CHALMERS


# Hierarchical Temporal Memory for Behavior Prediction

*Master's Thesis in Intelligent Systems Design*

DAVID BJÖRKMAN

Department of Applied Information Technology
Chalmers University of Technology
Gothenburg, Sweden 2011
Report No. 2011:081
ISSN: 1651-4769

**Summary**

This thesis is about researching and analyzing Hierarchical Temporal Memory, specifically the newly developed "HTM Cortical learning algorithms"[3] developed by Jeff Hawkins and the company Numenta. Two problems are addressed. Can this type of hierarchical memory system make an internal representation of simple data sequences at the input? And if so, does it take long to learn? Two C++ applications were developed in this thesis. The first program is used to analyze the algorithm, and the second program is used to visualize the internal states of the network. The results is very dependent of how the system is configured. If enough resources are available, the system can learn sequences, and it does not take long for the system to learn.

*Keywords:* HTM: Hierarchical Temporal Memory

**Acknowledgments**

I would like to thank my supervisor at Bitsim Kalle Lindbeck, my supervisor at Chalmers Claes Strannegård, and my parents Mats and Anna-Lena Björkman for their support.

Göteborg, January 2, 2012
David Björkman

# Contents


**1 Introduction** **1**

1.1 Background . . . 1

1.2 Purpose . . . 1

1.3 Motivation . . . 1

1.4 Problem specification . . . 1

1.5 Method . . . 2

**2 Hierarchical Temporal Memory: An overview** **3**

2.1 Background . . . 3

2.2 The Spatial Pooler . . . 3

2.3 The Temporal Pooler . . . 4

2.4 The result of combining spatial and temporal poolers . . . 4

2.5 The Hierarchy . . . 5

**3 Implementation details** **6**

3.1 SpatialPooler(): . . . 6

3.2 TemporalPooler(): . . . 7

**4 Temporal learning: One layer** **9**

4.1 Set up . . . 9

4.2 Analysis . . . 10

**5 Tweaking of parameters** **14**

5.1 Shared: . . . 14

5.2 Spatial Pooler: . . . 14

5.3 Temporal Pooler: . . . 15

**6 Results** **16**

6.1 Result . . . 16

6.2 Conclusion . . . 16

**7 Discussion/Future work** **17**

7.1 HTM as a novelty detector . . . . . 17

7.2 HTM as a predictor . . . . . 17

7.3 HTM for hardware implementation . . . . . 17

**References** **19**

**Appendices** **20**

**A HTM Code** **20**

**B Qt application output** **37**

# 1 Introduction

## 1.1 Background

In many AI applications, a specific algorithm will work with only one type of problem. In order to build more powerful systems that can classify many different types of sensory data a new approach is needed. There is one biological system known for this type of capability: the mammalian brain, or more specifically the neocortex. There has not been a unified theory of how the neocortex works as a whole, until a book was released on the subject called On Intelligence[2] written by Jeff Hawkins and Sandra Blakeslee, published in 2004. In it Hawkins lays the foundation for a technology called Hierarchical temporal memory, abbreviated HTM. Hawkins has sins started a company called Numenta to develops this technology to make it possible to build what he calls "truly intelligent machines".

If a system is intelligent or not has previously been defined by its behavior i.e. it must act in an intelligent way to be intelligent. Hawkins definition of intelligence is by prediction. He states that if a system constantly makes predictions, and matches incoming data to those predictions in order to make new more refined predictions, it can be called intelligent.
This is a system that constantly strives for a better understanding of what is going on. A system that continuously learns and updates its inner model of the world around it.

## 1.2 Purpose

The purpose of this theses is researching and analyzing Hierarchical Temporal Memory, specifically the newly developed "HTM Cortical learning algorithms"[3] developed by Numenta.

## 1.3 Motivation

In the search for a good algorithm for intelligent systems I had three important criteria.

- Time series functionality: The algorithm must be able to recognize sequences.
- Biological plausible: The algorithm should not be based on complex mathematics, but based on simple rules.
- Generality: The algorithm should not care what kind of data it is classifying.

The motivation for this thesis is understanding how these algorithms work. Therefor this thesis will be good for people who want to learn more about htm's.

## 1.4 Problem specification

- Can the system make an internal representation of the data sequence at the input?
- Does it take long for the system to make such a representation?

### 1.5 Method

In order to analyze Numentas HTMs, the algorithm, or the "HTM Cortical learning Algorithms" as it is called, was implemented in C++. The code was written from pseudo code found in [3], on Numentas web page [1] and can be found in appendix A. The analysis is limited to one layer of HTM, learning a sequence of zeros and ones. A Qt application was also developed for visualization. Output prom the Qt application can be seen in appendix B.

# 2 Hierarchical Temporal Memory: An overview

## 2.1 Background

HTM is based on a theory of how the different regions of the neocortex uses a common algorithm to make sense of the world by constantly predicting what might happen next and in the future.

Hierarchical temporal memory is modeled after the human brains intuitive ability to discover and predict patterns. HTM regions are connected in an hierarchy. Each region consists of two parts. The Spatial Pooler and the Temporal Pooler. In the examples below HTM is applied to image analysis, although HTMs can be applied to any input space.

## 2.2 The Spatial Pooler

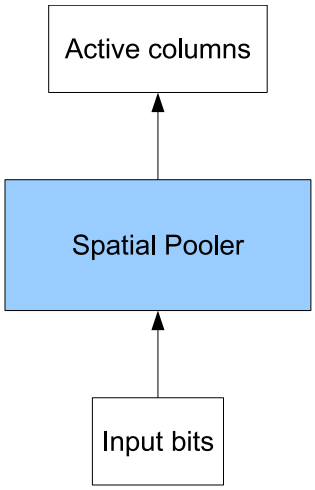


**Figure 2.1:** *The spatial poolers input is the input bits to all the columns in the region.*

The Spatial Pooler creates a sparse distributed representation of its inputs. The spatial pooler learns to represent a pattern in the best possible way with it's given resources. Even if an image is noisy or warped, it will roughly give the same output to the temporal pooler. The spatial poolers output to the temporal pooler is a set of active columns.

## 2.3 The Temporal Pooler

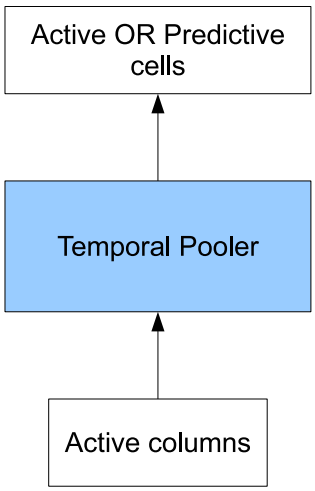


**Figure 2.2:** *The temporal poolers output is the combination of what is currently occurring and what is predicted.*

The Temporal pooler tries to predict what input will come next. Each column has a number of associated cells. The cells in a column represent different temporal contexts that the column can be a part of. If two particular columns are activated in sequence again and again, one of the second columns cells will create a connection between itself and a cell in the other column. By doing this, the first sell remembers the sequence and can predict when it will be active next.

A cell can be in a predictive state and/or in a active state. A cell enters the predictive state if it expects to be active in the following time step. A cell enters the active state if it was in the predictive state the last time step. If no cell was predicted the last time step (no connections), all cells in the column will enter the active state to show that the input was not recognized.

The output from the temporal pooler is the logical OR of the active and predictive state of each cell in all columns. These zeros and ones are then the input to the spatial pooler in the next higher layer in the hierarchy.

## 2.4 The result of combining spatial and temporal poolers

The result is a very robust noise tolerant system that dynamically will adapt to its environment. Like the human brain it is robust and failure tolerant. Because HTMs uses sparse distributed patterns, it can lose much of its resources (cells, columns), and still be able to make the exact same predictions. The connections between cells is dynamic and is constantly adapting to represent what the HTM is exposed to.

## 2.5 The Hierarchy

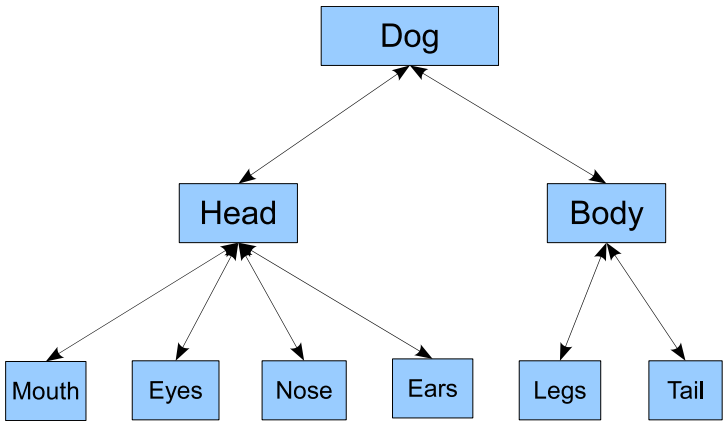


**Figure 2.3:** *The representation of a dog stored hierarchically.*

The output from each level in the hierarchy is more stable than its input. This will result in slower changing pattern at the top, and faster changing patterns at the bottom. A typical low level concept is for example the pixels in an image that change color from frame to frame in a video. At this level nothing can be said of the overall events in the video, but it is vital in creating a higher understanding of events. A typical high level in the hierarchy can for example represent a running dog. lower in the hierarchy is all the parts that a dog generally consists of, for example legs, a head and a tail. At the high level nothing can be said about the rapid color changes of the incoming pixels, but it can easily distinguish if the dog in the i video is running or standing still.

# 3 Implementation details

Two application was created to test the properties of HTMs. To analyze a single level in the hierarchy a Console application written in C++ was sufficient. To visualize of many layers put together in an hierarchy a more powerful visualization tool was needed. Qt offered a fast way to create a GUI Application that lets the user experiment with the algorithm already coded in C++. While this analysis will be about only one layer of HTM, output images from the Qt application is presented in appendix B.
Both applications were coded with readability and not performance in mind.

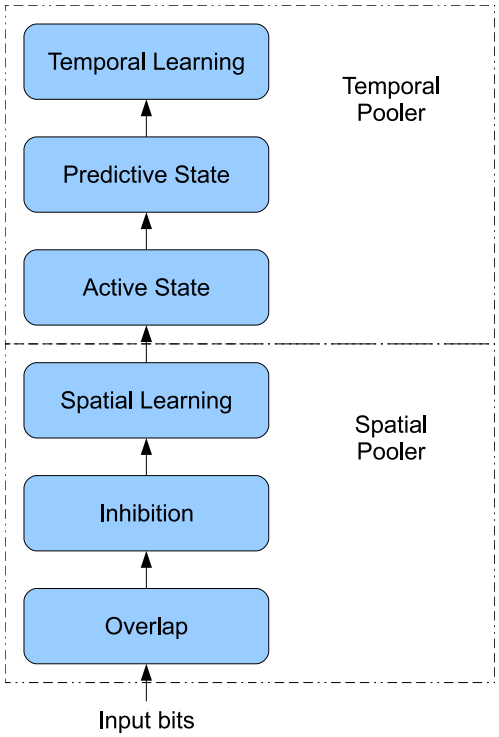


**Figure 3.1:** *The sequential flow from input to output in the implementation.*

The main Class is called HTM, and consists of all functions needed for a Hierarchy of HTM regions to classify its inputs over time. Each time step, the function "HtmTimeStep" is called. It keeps track of all the levels/regions and calls the functions "SpatialPooler" and "TemporalPooler" for each.

## 3.1 SpatialPooler():

The Spatial Pooler Function operates on all the columns on a specific level and has the main task of calling the functions "overlap", "inhibition" and "spatialLearning". The input to the spatial pooler is an array of zeros and ones with a length of "numInputBits"( see chapter Tweaking of parameters). This input can come from sensors or a region lower down in the hierarchy. The input bits are randomly selected to be included in a list of potential synapses (potential connections) for each column. It is this randomization that creates the distribution of what columns that

can be activated by which input bits. Each synapse has a permanence value associated with it. This value is a metric of how permanent or constant the connection is. If this value is above a threshold, the synapse is considered connected and can transfer information from an input bit to the column.

**overlap():**
Calculates how much overlap (how many of its connected synapses that have active input) each column has. There is a threshold which has to be exceeded in order for a column to count its overlap. The overlap value for each column is also multiplied by a boost value.

**inhibition():**
Sets the columns, within a neighborhood, with the highest overlap as active. It is this function that creates the sparseness of the active columns, so that only a small percentage is active at any given time.

**spatialLearning():**
Increases the permanence (i.e how constant a connection is) of the synapses (connections) with the input one, and decreases it for synapses with the input zero. This is done for active columns only. It also increases the boost value for columns that has a low activation rate compared to its neighbors, and increases the permanence values for synapses of columns that frequently has overlap values below the threshold to be counted. Lastly the function calculates a new neighborhood size for the layer. The size of the neighborhood affects how sparse the sparse distributed representation of activated columns get.

### 3.2 TemporalPooler():

Some new concepts needs to be introduced in order to understand the temporal pooler. The spatial pooler operates on columns, and the temporal pooler operates on the cells within a column. Each cell can send distal dendrite segments (short: segment) far away form its column and form synapses with other cells in other columns. There are two types of segments. Normal segments, just referred to as segments, and sequence segments. If any of a cells segments, sequence or not, is activated the cell enters the predictive state. The sequence segments are segments that is formed to represent a transition in a sequence of column activations. If a cell in column has an active sequence segment, the cell can be expected to enter the active state next time step.The non sequence segments are added to give the output extra stability. They does not try to specifically predict what will happen in the next time step, but sometime in the future. The segments and their synapses are what creates the temporal context for a particular column, and it is these segments and synapses that are trained and constantly adapting. In the spatial pooler there are only one segment per column with multiple synapses, one for each input bit from below. In the temporal pooler on the other hand, there are many segments per cell with multiple synapses on each segment.

The Temporal Pooler Function has the main task of calling the functions "calcActiveState", "calcPredictiveState" and "temporalLearning".

**calcActiveState():**
CalcActiveState operates on cells in active columns only, and has two main tasks.
**Choosing active cells:** Choose which cells (in an active column selected by the Spatial Pooler) that should become active. There are two scenarios. The first one is if any cell predicted its activation with a sequence segment in the previous time step. If this is the case, those cells will become active. The second scenario is that no cells was in the predictive state in the previous time step, in which case all cells in the column will become active.

**Choosing learn cell:** Choosing which single cell in the column will be in the learn-state for this time step.The learn-state is an internal state that allows a cell to form synapses with cells of other columns if it is not already part of a learned sequence. There are two ways of selection. The first one is if a cells predictive state was activated by a sequence segment, that was activated by a cell in the learn-state, in the last time step. That cell will then be chosen as the learn-cell. If no cell was selected this way, the second way of selection is utilized, in which the best cell gets to be learn-cell as follows.

For each cell the best segment is determined. The best segment is the segment which had the largest number of active synapses during the previous time step. The cell is chosen that had the highest number of active synapses on its best segment. If there was no active synapses on any segment, then the cell with the fewest number of segments is chosen as learn-cell.

When the learn-cell is chosen the second way, it forms synapses with cells that was active in the previous time step. Those synapses are added on the cells best segment. If there were no segment with active synapses, a new segment is created to add the synapses to. Segments created this way are sequence segments, segments trying to predict a cells activation in the next time step.

**calcPredictiveState():**
CalcPredictiveState operates on cells in all columns, active or not, and has three tasks.
**Choosing predictive cells:** A cell will enter the predictive state if enough of the synapses on any of its segments are active.
**Reinforce synapses:** All synapses on segments that has enough active synapses are reinforced. When a reinforcement is carried out, the permanence values of the selected synapses are increased.
**Make synapses further back in time:** These synapses are added to the best segment (the segment that had the most active synapses) the previous time step. If no segments are found, a new segment is created to add the synapses to. Segments created this way are non sequence segments, segments trying to predict a cells activation some time in the future.

**temporalLearning():** This function executes requests for changes to segments and synapses that was made in both "calcActiveState" and "calcPredictiveState". If it is requested to add synapses or segments, they are added here. If the segment under consideration already exists one of two things can happen.
**Positive reinforcement:** All positive changes to synapses permanence values are put in a queue until the cell associated with the changes becomes the learn cell.
**Negative reinforcement:** If a cell is not the learn cell and was in the predictive state last time step but not in this one, then the queued synapses gets their permanence values decremented.

# 4 Temporal learning: One layer

In this section one layer of HTM, running the temporal pooler, will be analyzed.

## 4.1 Set up

The cortical learning algorithms can be set up in many ways to achieve completely different results. Before analyzing the temporal pooler, or any other algorithm with unknown behavior, there are three important things needs to be considered.

**Reproducibility**
The output must be consistent when giving the same input twice. However in this case when analyzing patterns through time, it is also important to be aware of that the the changes of the output is dependent of not just the current input, but prior input as well. To achieve this reproducibility it is important for the researcher to have an clear overview of both the input and output of the system.

As HTM's are developed for large sparse distributed data streams from the real world, it is easy to see that it is hard to get a clear overview, without reducing the size of the system analyzed.

When reducing the size, one thing must be absolutely certain. It is whether the algorithm will work in the same way in a small versus a large system. From the view of the temporal pooler this is true. The amount of cells per column can be low, and they will still function the same way. For the spatial pooler though it is more complex. When reducing the number of columns for a layer, the representations stored gets simpler, e.g. a recognized object is involving 5 active columns in stead of fifty. This may be a problem for large systems, and systems with high amounts noise, but to analyze the temporal characteristics of the algorithm, a small system will suffice.

The simplest possible test case is to look at a stream of zeros and ones just one bit wide. The easiest then would be to have just one column, but there is one important thing to remember about the spatial pooler. When calculating the overlap of each column, the algorithm counts the number of active synapses it is connected to. This has the implication that when just one bit is used and a zero is the input to the spatial pooler, the column will not be active and the temporal pooler will not have anything to do. The solution for the simplest possible system is then to have two columns, the first active when the input bit is one, the other active when the input bit is zero.

**Informative Output**
Fore these tests the HTM is set up as a novelty detector. This means that that the output will be one when any of the columns are surprised of its activation i.e. no cell in the column was in the predictive state the previous time step. When studying surprised columns, two interesting things can be observed. When in a sequence new synapses are needed to update the internal model, and how long it takes to memorize an unknown sequence for the first time.

**Algorithm set up**
In order to test the temporal abilities of the HTM region, certain parts of the algorithm was temporarily changed or even switched off. The potential synapses that connects the input bits to the columns was fixed in order to make the same columns active for a specific input, each time the test ran. Spatial learning was turned off to make the activation of columns stable over the course of the test.

In the temporal pooler, each cell is only allowed to make predictions one step ahead in the future. This makes it easier to analyze the temporal properties of the region, although the mechanism to predict further in the future is an integral and important part of the algorithm, as it provides the output with more stability, each time a temporal pattern is repeated. Lastly all segments with no active synapses are deleted at the end of each time step, to make the program run faster.

The input sequence used for these tests are twenty bits long and sufficiently complex in order to observe how learning occurs. There are three mechanics to extend and reduce the memory:

**Number of cells per column**
**Permanence decrementation value**
**Number of synapses per segment**

## 4.2 Analysis

There are two aspects that will be analyzed. When during a specific sequence will the network be surprised, and when will it correct or extend its connections to increase its understanding of its inputs. The latter will happen even though the column is not surprised. The red in the tables below denotes when in the input sequence the HTM becomes surprised and creates new synapses

**Test 1:**
Cells per column: 3
Permanence dec: 0
Synapses per segment: 3

| Iteration | Input pattern | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 4.1:** *It takes only one iteration to remember the pattern*

**Test 2:**
Cells per column: 3
Permanence dec: 0
Synapses per segment: 2

| Iteration | Input pattern | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 4.2:** *Decreasing synapses per segment to two slows the learning process down*

**Test 3:**
Cells per column: 3
Permanence dec: 0
Synapses per segment: 1

| Iteration | Input pattern | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 8 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 9 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 4.3:** *When synapses per segment is decreased to one, the pattern cannot be fully learned*

**Test 4:**
Cells per column: 3
Permanence dec: 0.007
Synapses per segment: 3

| Iteration | Input pattern | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 4.4:** *When Making Permanence dec non zero the system will start forgetting learned connections*

**Test 5:**
Cells per column: 3
Permanence dec: 0.01
Synapses per segment: 3

| Iteration | Input pattern | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 8 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 9 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 4.5:** *When Making Permanence dec bigger (0.01 in this case), the system is forgetting so fast that certain parts of the sequence must be re-learned each iteration*

**Test 6:**
Cells per column: 2
Permanence dec: 0
Synapses per segment: 3

| Iteration | Input pattern | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 8 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 9 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 4.6:** *Decreasing the number of cells per column to two will force the system to represent its inputs in simpler context*

**Test 7:**
Cells per column: 1
Permanence dec: 0
Synapses per segment: 3

| Iteration | Input pattern | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 4 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 6 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 7 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 8 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 9 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 10 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

**Table 4.7:** *When there is only one cell per column there are only one context per column that can be learned (in this case 01 and 10 because it is the first context in the sequence)*

# 5 Tweaking of parameters

The parameters can be divided in groups according to what part(s) of the algorithm that uses them.

## 5.1 Shared:

These parameters are used by both the spatial and temporal poolers, because they apply to all synapses. If he permanence value for a synapse is greater than **connectedPerm**, the synapse becomes connected and can transfer information. If this value is constant the other parameters in this category can be tweaked to achieve different things. **InitialPerm** is the initial permanence value assigned to a newly created synapse. If this value is close to the value of connectedPerm it does not take much training to ether exceed or fall below connectedPerm. InitialPerm is typically larger than connectedPerm, but how much depends on how hard it should be for a newly formed synapse to become active. **PermanenceInc** controls how fast a synapse's permanence value can be increased, as the value is added each time a synapse is reinforced. The same procedure is also true for **permanenceDec**, but instead of controlling the reinforcement, it controls how fast the permanences values can be decreased. PermanenceDec determines how fast a HTM network forgets learned connections. The last parameter **iterations** controls how many previous time steps that is used to calculate the average activity for a column in the spatial pooler. Decreasing this parameter will make the columns forget past activations faster, and the system will be more focused on more recent activity.

## 5.2 Spatial Pooler:

**NumColumns** controls how much memory is allocated to the spatial pooler, i.e. how large the sparse distributed representations the spatial pooler will have. More columns will result in a more robust and redundant system, as many columns will be part of the representation of any one thing.
**MinOverlap** dictates the minimum number of active synapses in a columns input to participate in the inhibition. A column with overlap below MinOverlap has no chance of being active.
**DesiredLocalActivity** is a number that controls how many columns that should be active within a neighborhood after inhibition.

### 5.3 Temporal Pooler:

These prameters are the specific to the temporal pooler.
**CellsPerColumn** determines how many different contexts a column can represent. Increasing the number of cells per column will allow the column to store longer and more complex patterns.
**MaxSynapesCount** indicates how many synapses there is room for on one segment. If the allowed amount of synapses on a segment is too low, only simple patterns can be stored. If the amount on the other hand is increased, more complex patterns can be learned if the number of cells per column is sufficiently large.
**ActivationThreshold** controls how many active synapses on a segment is needed for the segment to be active and make a cell enter the predictive state. If activationThreshold is bigger, it takes more active cells to activate the segment.
**MinThreshold**: If a segment has fewer active synapses than MinThreshold, it cannot be chosen as the best segment.

# 6 Results

## 6.1 Result

The results of the tests carried out is highly dependent on how the system is configured. The following was noted during the tests:

- Decreasing synapses per segment slows the learning process down.
- When decreasing the number of synapses per segment too much, the pattern cannot be fully learned.
- When Making Permanence dec non zero the system will start forgetting learned connections.
- When Making Permanence dec too big, the system is forgetting so fast that certain parts of the sequence must be re-learned each iteration.
- Decreasing the number of cells per column will force the system to represent its inputs in simpler context.
- When there is only one cell per column there are only one context per column that can be learned.

## 6.2 Conclusion

When there are a sufficiently large number of cells per column, many synapses per segment, and no to little permanence decremantation (forgetting of connections), then the system is a really fast learner. However when limits are imposed, the system still learns to the best of its ability. In terms of the problem specification the Conclusions can be described:

- Yes the HTM can make an internal representation of the data sequence at the input.
- No it does not take long for the system to make such a representation if enough resources are assigned.

# 7 Discussion/Future work

This chapter will cover what is needed to use HTM's in its current form in real world applications.

## 7.1 HTM as a novelty detector

As shown in the analysis section of this thesis, an HTM can be used to find out when something unexpected happens in an incoming stream of data. This is of value in many applications where it is critical that things stay the same. If for example the temperature in a building is critical, an HTM can for example sense a fluctuation in the outside temperature long before it can be measured on the inside of the building. In such an example the HTM would have sensor input form both inside and outside of the building. If everything stays the same the HTM would do nothing, but if a difference was found i.e. the HTM became surprised on some level, an alarm would go of.

## 7.2 HTM as a predictor

In its current state a cell don't make a distinction between predicting the next and future events. To specifically access the prediction for the next time step, it is the sequence segments that are useful to look at, as they carry information of what cells were active in the previous time step.

## 7.3 HTM for hardware implementation

The first thing to remember about HTM's systems is that they operate mainly in parallel. Each cell receives its input and produces its output in parallel to the other cells in the same layer. The code written in this thesis runs on one processor. This is not ideal, as this processor has to loop through cells and compute their output sequentially, which takes time.

Even though there can be many cells to loop through, there are many more synapses to keep track of. As each cell learns it creates a number segments. These segments has in turn many synapses each, and each synapse has a floating point permanence value. The system learns through making connections between cells, so more experience and accuracy means more and more synapses. The amount of memory is linearly increasing with the number of synapses the system uses.

Memory usage is also affected by how many time steps back in time that states (active, predictive and learn-states) for each cell are stored. The memory usage can only be reduced by tweaking the appropriate parameters to fit the problem the HTM will operate on. Execution time though, can be increased by for example run the HTM code on a multi core processor or a computer cluster.

One step further in the speedup direction would be to simply implement the algorithm in an FPGA. By using an FPGA for implementation of the HTM the speed can be significantly increased, but the the large amount of resources needed to keep track of the connections required can still cause problems. The currency for FPGA:s is logic-cells. A logic-cell in an FPGA is used to perform a boolean operation of some kind. all these operations can be run in parallel, which

fits HTM:s perfectly. One aspect of FPGAs that could be used is dynamic reconfiguration. This could be used to mimic how a HTM system makes or removes connections between cells.

To use an FPGA or ASIC to its full potential the algorithm must be implemented in concurrent code. To make this conversion of the HTM code, the first step would be to find the parts of the code that can be run in parallel.

A column must work as an autonomous unit. There can't be any overarching control program for all the columns, a column must be able to work independently to its neighbors. To achieve this it is the competition between columns, described in the spatial poolers inhibition function, must work on a column by column basis, but for obvious reasons have connections to neighboring columns. In the temporal pooler there is similar requirement. The cells within a column need access to many cells in neighboring columns. The bottleneck will not be the number of neurons or speed that can be achieved, it will be connectivity.

## References

[1] Numenta. URL www.numenta.com.

[2] Sandra Blakeslee Jeff Hawkins. *On Intelligence*. Times Books, a division of Henry Holt and Company, 2004.

[3] Inc. Numenta. Hierarchical temporal memory including htm cortical learning algorithms, 2011.

# Appendices

## A HTM Code

```
/*Numenta License for Non-Commercial Use
Copyright (c) 2011 David Bjorkman

All rights reserved.

This software is based upon or includes certain works and algorithms related
to hierarchical temporal memory ("HTM") technology published by Numenta Inc.
Numenta holds patent rights related to HTM and the algorithms used in this
software.  Numenta has agreed not to assert its patent rights against
development or use of independent HTM systems, as long as such development
or use is for research purposes only, and not for any commercial or production
use.  Any commercial or production use of HTM technology that infringes on
Numenta's patents will require a commercial license from Numenta.

Based on the foregoing, this software is licensed under the terms below, for
research purposes only and not for any commercial or production use.  For
purposes of this license, "commercial or production use" includes training an
HTM network with the intent of later deploying the trained network or
application for commercial or production purposes, and using or permitting
others to use the output from HTM technology for commercial or production
purposes.

Redistribution and use in source and binary forms, with or without
modification, are permitted (subject to the limitations in the disclaimer
below) provided that the following conditions are met:

* Redistributions of source code, including any modifications or derivative
works, must retain the full text of this license and the following disclaimer,
and be subject to the terms and conditions of this license.

* Redistributions in binary form, including any modifications or derivative
works, must reproduce the full text of this license and the following
disclaimer in the documentation and/or other materials provided with the
distribution, and be subject to the terms and conditions of this license.

* Neither the name of Numenta, Inc., David Bjorkman, nor the names of other
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.

NO EXPRESS OR IMPLIED LICENSES TO ANY PATENT RIGHTS ARE GRANTED BY THIS
LICENSE.  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.*/

#include <vector>
class HTM
{
public:
    HTM(int numInputBits, int numColumns, unsigned numCells, float permDec, int maxSyn);
    ~HTM();
```

```
    void SpatialPooler(unsigned t, std::vector<int> input);
    void TemporalPooler(unsigned t);
    std::vector<std::vector<int>> activeColumns;

private:
    //structures
    struct synapse
    {
        float permanence;       //permanence value
        int sourceInput;        //index to region input
    };
    struct syn
    {
        float permanence;       //permanence value
        int column_index;       //index to a column
        int cell_index;         //index to a cell
    };
    struct synapseUpdate
    {
        float permanence;       //permanence value
        int column_index;       //index to a column
        int cell_index;         //index to a cell
        int synapse_index;
    };
    struct segmentUpdate
    {
        int segIndex;
        bool sequenceSegment;
        std::vector<synapseUpdate>activeSynapses;
    };
    struct segment
    {
        float activity;
        bool sequenceSegment;
        unsigned num_active_synapses;
        std::vector<syn>potentialSynapses;
        std::vector<syn>connectedSynapses;
    };
    struct cell
    {
        int predictSegmentIndex;
        std::vector<bool>predictiveState;
        std::vector<bool>learnState;
        std::vector<bool>activeState;
        std::vector<segment>segment;
    };
    struct  column
    {
        std::vector<float> overlap;
        float boost;
        float activeDutyCycle;
        float minDutyCycle;
        float overlapDutyCycle;
        int lastLearnCell;
        std::vector<synapse>potentialSynapses;
        std::vector<synapse>connectedSynapses;
        std::vector<int>neighbors;

        std::vector<cell>cell;
        std::vector<bool>surpriseState;
    };
    //Constants
    unsigned iterations;
    float minOverlap;
    float connectedPerm;
    float permanenceInc;
```

```
    float permanenceDec;
    int desiredLocalActivity;
    float inhibitionRadius;
    unsigned cellsPerColumn;
    int activationThreshold;
    int minThreshold;
    int maxSynapseCount;
    float initialPerm;
    enum state {activeState,learnState};
    //Variables
    std::vector<std::vector<int>> mInput;
    std::vector<std::vector<std::vector<segmentUpdate>>> segmentUpdateList;

    /*Functions Spatial Pooler*/
    void overlap(unsigned t);
    void inhibition(unsigned t);
    float kthScore( std::vector<int>cols, int k, unsigned t);
    void spatialLearning(unsigned int t);
    float maxDutyCycle(int i);
    float updateActiveDutyCycle(int c, unsigned t);
    float boostFunction(int c);
    float updateOverlapDutyCycle(int c, unsigned t);
    void increasePermanences(int c, float permInc);
    float avarageReceptiveFieldsize();

    /*Helper functions for Spatial Pooler*/
    void updateConnectedSynapses();
    void updateNeighbors();
    float computeDistance(int i, int j);
    int randInt(int low, int high);
    float randFloat(float low, float high);

    /*Functions Temporal Pooler*/
    void calcActiveState(unsigned t);
    void calcPredictiveState(unsigned t);
    void temporalLearning(unsigned t);
    int getActiveSegment(int c, int i, int t, state cellState);
    bool segmentActive(int c, int i, int s, int t, state cellState);
    int getBestMatchingCell(unsigned c, unsigned t, unsigned wrongPredCell, int& seg);
    int getBestMatchingSegment(int c, int i, int t);
    segmentUpdate getSegmentActiveSynapses(int c,int i,int s,int t,bool newSynapses=↩
        false);
    void adaptSegments(int c, int i, bool positiveReinforcement);

    /*Helper functions for Temporal Pooler*/
    float getSegmentActivity(int c, int i, int s, int t, state cellState);
    void updateCellSynapses(int c, int i);

public: void initTimeStep();
public: std::vector<HTM::column> col;
};
```

```
/*Numenta License for Non-Commercial Use
Copyright (c) 2011 David Bjorkman

All rights reserved.

This software is based upon or includes certain works and algorithms related
to hierarchical temporal memory ("HTM") technology published by Numenta Inc.
Numenta holds patent rights related to HTM and the algorithms used in this
software.  Numenta has agreed not to assert its patent rights against
development or use of independent HTM systems, as long as such development
or use is for research purposes only, and not for any commercial or production
use.  Any commercial or production use of HTM technology that infringes on
Numenta's patents will require a commercial license from Numenta.

Based on the foregoing, this software is licensed under the terms below, for
research purposes only and not for any commercial or production use.  For
purposes of this license, "commercial or production use" includes training an
HTM network with the intent of later deploying the trained network or
application for commercial or production purposes, and using or permitting
others to use the output from HTM technology for commercial or production
purposes.

Redistribution and use in source and binary forms, with or without
modification, are permitted (subject to the limitations in the disclaimer
below) provided that the following conditions are met:

* Redistributions of source code, including any modifications or derivative
works, must retain the full text of this license and the following disclaimer,
and be subject to the terms and conditions of this license.

* Redistributions in binary form, including any modifications or derivative
works, must reproduce the full text of this license and the following
disclaimer in the documentation and/or other materials provided with the
distribution, and be subject to the terms and conditions of this license.

* Neither the name of Numenta, Inc., David Bjorkman, nor the names of other
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.

NO EXPRESS OR IMPLIED LICENSES TO ANY PATENT RIGHTS ARE GRANTED BY THIS
LICENSE.  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.*/

#include "HTM.h"
#include "time.h"
using namespace std;
HTM::HTM(int numBits, int numColumns, unsigned numCells, float permDec, int maxSyn)
{
    srand ( (unsigned int) time(NULL) ); //Get seed for randomizer
    connectedPerm=0.2f;
    minOverlap = 1;
    permanenceInc=0.01f;
    permanenceDec=permDec;
    desiredLocalActivity = 2;
    iterations = 5;
    cellsPerColumn = numCells;
    activationThreshold = 0;
    minThreshold = 0;
```

```
    maxSynapseCount = maxSyn;
    initialPerm =0.21f;
    column col_tmp;
    synapse syn_tmp;
    cell cell_tmp;
    for (int c=0; c < numColumns; c++)
    {
        col.push_back(col_tmp);
        col[c].overlapDutyCycle = 0;
        col[c].activeDutyCycle=0;
        //potentialSynapses is a random selection of input bits
        int j=0;
        /*for (int i = 0; i < numBits; i++)
        {
            if (randInt(0,1))
            {
                syn_tmp.sourceInput=i;
                col[c].potentialSynapses.push_back(syn_tmp);
            }
        }*/

        /*Special Case For Testing*/
        if(c < numColumns/2)
        {
            syn_tmp.sourceInput=0;
            col[c].potentialSynapses.push_back(syn_tmp);
        }
        else
        {
            syn_tmp.sourceInput=1;
            col[c].potentialSynapses.push_back(syn_tmp);
        }
        /*////////////////////////////////////////////*/

        for (unsigned int s=0; s < col[c].potentialSynapses.size(); s++)
        {
            //permanence values range from 0.0 to 1.0
            float min = connectedPerm*1.1f;//*0.9f;
            float max = connectedPerm*1.1f;
            col[c].potentialSynapses[s].permanence = randFloat(min,max);
        }
        col[c].boost = 1;
        std::vector<std::vector<segmentUpdate>> segmentUpdate_tmp;
        std::vector<segmentUpdate> segment_tmp;
        segmentUpdateList.push_back(segmentUpdate_tmp);
        for (unsigned i = 0; i < cellsPerColumn; i++)
        {
            col[c].cell.push_back(cell_tmp);
            segmentUpdateList[c].push_back(segment_tmp);
        }
    }
    updateConnectedSynapses();                          //ConnectedSynapses depends on ↩
        the permanence value on the synapses
    inhibitionRadius = avarageReceptiveFieldsize();     //Affects wish columns are ↩
        neighbors
    updateNeighbors();                                  //Calculates neighbors for each ↩
        column
}
HTM::~HTM(){}
/*Spatial Pooler*/
void HTM::SpatialPooler(unsigned t, std::vector<int> input)
{
    mInput.push_back(input);
    overlap(t);
    inhibition(t);
    //spatialLearning(t);
```

```
    //Update stuff!!!
    updateConnectedSynapses();
    updateNeighbors();
}
void HTM::overlap(unsigned t)
{
    for (unsigned c=0; c < col.size(); c++)
    {
        col[c].overlap.push_back(0);
        for (unsigned s=0; s < col[c].connectedSynapses.size(); s++) //for every ↩
            connected synapse
        {
            col[c].overlap[t] = col[c].overlap[t] + mInput[t][col[c].connectedSynapses[s↩
                ].sourceInput];
        }
        if (col[c].overlap[t] < minOverlap)
        {
            col[c].overlap[t] = 0;
        }
        else
        {
            col[c].overlap[t] = col[c].overlap[t] * col[c].boost;
        }
    }
}
void HTM::inhibition(unsigned t)
{
    std::vector<int>tmp;
    float minLocalActivity = 0;
    activeColumns.push_back(tmp);
    activeColumns[t].clear();
    for (unsigned int c=0; c < col.size(); c++)
    {
        minLocalActivity = kthScore(col[c].neighbors, desiredLocalActivity, t);
        if (col[c].overlap[t] > 0 && col[c].overlap[t] >= minLocalActivity)
        {
            activeColumns[t].push_back(c);      //Extend activeColumns with column c
        }
    }
}
float HTM::kthScore( std::vector<int>cols, int k, unsigned t)
{
    //Returns the k'th highest overlap value
    if (cols.size() != 0)
    {
        int index=cols[0];
        std::vector<column>tmp = col;
        std::vector<float>v;
        for (int i = 0; i < k; i++)
        {
            for (unsigned int c = 0; c < cols.size(); c++)
            {
                if (tmp[cols[c]].overlap[t] >= tmp[index].overlap[t])
                {
                    index = cols[c];
                }
            }
            v.push_back(tmp[index].overlap[t]);
            tmp[index].overlap[t]=0;
        }
        return v[k-1];
    }
    else
    {
        return 0;
    }
```

```
}
void HTM::spatialLearning(unsigned t)
{
    for (unsigned i=0; i < activeColumns[t].size(); i++)
    {
        int c = activeColumns[t][i]; //Active Column index
        for (unsigned s=0; s < col[c].potentialSynapses.size(); s++)
        {
            if(mInput[t][col[c].potentialSynapses[s].sourceInput] == 1)
            {
                col[c].potentialSynapses[s].permanence = col[c].potentialSynapses[s].↩
                    permanence + permanenceInc;
                col[c].potentialSynapses[s].permanence = min(1.0f , col[c].↩
                    potentialSynapses[s].permanence);
            }
            else
            {
                col[c].potentialSynapses[s].permanence = col[c].potentialSynapses[s].↩
                    permanence - permanenceDec;
                col[c].potentialSynapses[s].permanence = max(0.0f , col[c].↩
                    potentialSynapses[s].permanence);
            }
        }
    }
    for (unsigned int c=0; c < col.size(); c++)
    {
        col[c].minDutyCycle = 0.01f * maxDutyCycle(c);
        col[c].activeDutyCycle = updateActiveDutyCycle(c,t);
        col[c].boost = boostFunction(c);

        col[c].overlapDutyCycle = updateOverlapDutyCycle(c,t);
        if(col[c].overlapDutyCycle < col[c].minDutyCycle)
        {
            increasePermanences(c, 0.1f * connectedPerm);
        }
    }
    inhibitionRadius = avaragereceptiveFieldsize();      //Affects wish columns are ↩
        neighbors
}
float HTM::maxDutyCycle(int i)
{
    std::vector<int> cols = col[i].neighbors;
    //Returns the maximum active duty cycle of the columns in cols
    float    max=0;
    for (unsigned int c = 0; c < cols.size(); c++)
    {
        if(col[cols[c]].activeDutyCycle > max)
        {
            max = col[cols[c]].activeDutyCycle;
        }
    }
    return max;
}
float HTM::updateActiveDutyCycle(int c, unsigned t)
{
    //Computes a moving average of how often column c has been
    //active after inhibition

    //Simple moving average:
    int acc=0;
    //unsigned iter = 1000;
    int size = min(iterations , activeColumns.size());
    int j = t;
    int limit = j - size;

    for(; j > limit; j--) //0-0,1-0,2-0....999-0,1000-1,1001-2...
```

```
    {
        for(unsigned i = 0; i < activeColumns[j].size(); i++)
        {
            if( activeColumns[j][i] == c)
            {
                acc++;
            }
        }
    }
    return float(acc)/iterations;
}
float HTM::boostFunction(int c)
{
    //Computes a boost value (>=1)for a column c.
    //If active is above min the boost value = 1.
    //The boost increases linearly once active starts falling below min

    float active = col[c].activeDutyCycle;
    float min = col[c].minDutyCycle;
    if(active > min)
    {
        return 1;
    }
    else if(active < min)
    {
        return (col[c].boost) * (1.5f);
    }
    else
    {
        return col[c].boost;
    }
    return 0;
}
float HTM::updateOverlapDutyCycle(int c, unsigned t)
{
    //Computes a moving average of how often column c has
    //overlap greater than minOverlap

    //Simple moving average:
    int newVal=0;
    int oldVal=0;
    if( col[c].overlap[t] > minOverlap)
    {
        newVal++;
    }
    if( t >= iterations && col[c].overlap[t-iterations] > minOverlap)
    {
        oldVal++;
    }
    return col[c].overlapDutyCycle + (float(newVal)/iterations) - (float(oldVal)/↩
        iterations);
}
void HTM::increasePermanences(int c, float permInc)
{
    for(unsigned s = 0; s < col[c].potentialSynapses.size(); s++)
    {
        col[c].potentialSynapses[s].permanence = col[c].potentialSynapses[s].permanence ↩
            * 1.0f + permInc;
    }
}
float HTM::avarageReceptiveFieldsize()
{
    float distance;
    float average = 0;
    for(unsigned c = 0; c < col.size(); c++)
    {
```

```
        distance = 0;
        for(unsigned s = 0; s < col[c].connectedSynapses.size(); s++)
        {
            int tmp = int(c) - col[c].connectedSynapses[s].sourceInput;
            tmp = tmp * tmp;
            distance += sqrt(float(tmp));
        }
        if(col[c].connectedSynapses.size() != 0)
        {
            distance /= col[c].connectedSynapses.size();  // Average for all connected ↩
                synapses for column s
        }
        else
        {
            distance = 0;
        }
        average += distance;
    }
    return average/col.size();      //Average for all columns
}
/*Helper functions for Spatial Pooler*/
void HTM::updateConnectedSynapses()
{

    for (unsigned int c = 0; c < col.size(); c++)
    {
        col[c].connectedSynapses.clear();
        for (unsigned int s = 0; s < col[c].potentialSynapses.size(); s++)
        {
            if(col[c].potentialSynapses[s].permanence > connectedPerm)
            {
                col[c].connectedSynapses.push_back(col[c].potentialSynapses[s]);
            }
        }
    }
}
void HTM::updateNeighbors()
{
    unsigned dMin, dMax;
    for (unsigned c = 0; c < col.size(); c++)
    {
        dMin = unsigned(max(0.0f, float(c) - inhibitionRadius));
        dMax = unsigned(min(float(col.size()) - 1.0f, inhibitionRadius));
        for (unsigned d = dMin; d < dMax; d++)
        {
            col[c].neighbors.push_back(d);
        }
    }
}
float HTM::computeDistance(int i, int j)
{
    return 0;
}
int HTM::randInt(int low, int high)
{
    int r = rand() % (high - low + 1) + low;
    return r;
}
float HTM::randFloat(float low, float high)
{
    float r = low + ( (high-low) * (float(rand())/RAND_MAX) );
    return r;
}
/*Temporal Pooler*/
void HTM::TemporalPooler(unsigned t)
{
```

```
    initTimeStep();
    calcActiveState(t);
    if (t>0)
    {
        calcPredictiveState(t);
        temporalLearning(t);
    }
}
void HTM::calcActiveState(unsigned t)
{
    //Sets the active state for each cell and chooses a learn cell
    int c,s,wrongPredCell;
    bool buPredicted, lcChosen;
    segmentUpdate sUpdate;

    for (unsigned j=0; j < activeColumns[t].size(); j++)
    {
        buPredicted = false;
        lcChosen = false;
        wrongPredCell = -1;
        c = activeColumns[t][j]; //Active Column index

        if (t>0)        //Is there a previous time-step?
        {
            for (unsigned i = 0; i < cellsPerColumn; i++)
            {
                if(col[c].cell[i].predictiveState[t-1] == true)
                {
                    //There must be an active segment!!!!!
                    s = getActiveSegment(c,i,t-1,activeState);                  //Get segment that was active due to activeState last time-step
                    if(col[c].cell[i].segment[s].sequenceSegment == true)   //Was it a sequence segment?
                    {
                        buPredicted = true;
                        col[c].cell[i].activeState[t] = true;                   //Set that cell to active
                        if(segmentActive(c,i,s,t-1,learnState))                 //Was the segment connected to a cell in learnState last time-step?
                        {
                            lcChosen = true;
                            col[c].cell[i].learnState[t] = true;                //Set that cell to learn state
                        }
                    }
                }
            }
        }
        if(buPredicted == false)    //No input was predicted last time step (no sequenceSegment)
        {
            for(unsigned i = 0; i < cellsPerColumn; i++)
            {
                col[c].cell[i].activeState[t] = true;       //Activate all cells
            }
            //col[c].surpriseState[t] = true;   //Surprised column
        }
        if(lcChosen == false)       //No learn cell was chosen
        {
            col[c].surpriseState[t] = true;     //Fix connections
            int s;
            int i = getBestMatchingCell(c,t-1,wrongPredCell,s);     //Choose best cell and best segment on that cell
            col[c].cell[i].learnState[t] = true;
            if (t>0)
            {
```

```
                sUpdate = getSegmentActiveSynapses(c,i,s,t-1,true);      //Add synapses ↩
                    to segment if there is any, otherwise create segment
                sUpdate.sequenceSegment = true;                          //Only sequence ↩
                    segments is added this way
                segmentUpdateList[c][i].push_back(sUpdate);
            }
        }
    }
}
void HTM::calcPredictiveState(unsigned t)
{
    //Sets the predictive state for each cell
    segmentUpdate activeUpdate, predUpdate;
    int predSegment;
    for (unsigned c = 0; c < col.size(); c++)
    {
        for (unsigned i = 0; i < cellsPerColumn; i++)
        {
            for (unsigned s = 0; s < col[c].cell[i].segment.size(); s++)
            {
                //For every active segment update the best segment for the cell
                if ( segmentActive(c,i,s,t,activeState) )
                {
                    col[c].cell[i].predictiveState[t] = true;
                    col[c].cell[i].predictSegmentIndex = s;

                    activeUpdate = getSegmentActiveSynapses(c,i,s,t,false);             ↩
                        //Old synapses on old segment
                    segmentUpdateList[c][i].push_back(activeUpdate);                    ↩
                        //No new segments, reinforce old ones!!!

                    /*Predict further in time*/
                    predSegment = getBestMatchingSegment(c,i,t-1);                      ↩
                        //Find the best segment last time step, or create a new one if ↩
                        none exists
                    predUpdate = getSegmentActiveSynapses(c,i,predSegment,t-1,true);    ↩
                        //New synapses to old (or new) segment
                    //segmentUpdateList[c][i].push_back(predUpdate);                    ↩
                        //Put the updates on the waiting list
                }
            }
        }
    }
}
void HTM::temporalLearning(unsigned t)
{
    //Learning: Adjusts the synapses of segments in the segmentUpdateList
    for (unsigned c = 0; c < col.size(); c++)
    {
        for (unsigned i = 0; i < cellsPerColumn; i++)
        {
            if (col[c].cell[i].learnState[t] == true)
            {
                adaptSegments(c,i,true);
                segmentUpdateList[c][i].clear();
            }
            else if (col[c].cell[i].predictiveState[t] == false && col[c].cell[i].↩
                predictiveState[t-1] == true)
            {
                //False prediction
                adaptSegments(c,i,false);
                segmentUpdateList[c][i].clear();
            }
            updateCellSynapses(c,i);
        }
    }
```

```
}
bool HTM::segmentActive(int c, int i, int s, int t, state cellState)
{
    //Returns true if the number of connected synapses on segment s that
    //are active due to cellState at time t is greater than activationThreshold

    int col_index;
    int cell_index;
    int count = 0;
    if (cellState == activeState)
    {
        for (unsigned j = 0; j < col[c].cell[i].segment[s].connectedSynapses.size(); j↩
            ++)
        {
            col_index = col[c].cell[i].segment[s].connectedSynapses[j].column_index;
            cell_index = col[c].cell[i].segment[s].connectedSynapses[j].cell_index;
            if (col[col_index].cell[cell_index].activeState[t] == true)
            {
                count++;
            }
        }
    }
    else if (cellState == learnState)
    {
        for (unsigned j = 0; j < col[c].cell[i].segment[s].connectedSynapses.size(); j↩
            ++)
        {
            col_index = col[c].cell[i].segment[s].connectedSynapses[j].column_index;
            cell_index = col[c].cell[i].segment[s].connectedSynapses[j].cell_index;
            if (col[col_index].cell[cell_index].learnState[t] == true)
            {
                count++;
            }
        }
    }

    if (count > activationThreshold)
    {
        return true;
    }
    else
    {
        return false;
    }
}
int HTM::getActiveSegment(int c,int i,int t, state cellState)
{
    /*************************************************************************/
    /*      For column c, cell i returns Segment index                       */
    /*      Get the best segment index in following order:                   */
    /*                                                                       */
    /*      Active                                                           */
    /*      Sequence segment                                                 */
    /*      Most active                                                      */
    /*************************************************************************/

    int activeCount = 0;
    int sequenceCount = 0;
    int activeIndex, sequenceIndex, moastActiveIndex;
    float activity = 0;
    float a;
    for (unsigned s = 0; s < col[c].cell[i].segment.size(); s++)
    {
        if(segmentActive(c,i,s,t,cellState))
        {
            activeCount++;
```

```
            activeIndex = s;
            if(col[c].cell[i].segment[s].sequenceSegment == true)
            {
                sequenceCount++;
                sequenceIndex = s;
            }
            a=getSegmentActivity(c,i,s,t,cellState);
            if (a>activity)
            {
                activity = a;
                moastActiveIndex = s;
            }
        }
    }
    if (activeCount==1)     //If just one segment is active return it's index
    {
        return activeIndex;
    }
    else if(activeCount > 1 && sequenceCount == 0)      //Else return the most active ↩
        segment's index
    {
        return moastActiveIndex;
    }
    else if(activeCount > 1 && sequenceCount == 1)          //If just one active segment↩
         is a sequenceSegment return it's index
    {
        return sequenceIndex;
    }
    else if(activeCount > 1 && sequenceCount > 1)       //Else return the most active ↩
        segment's index
    {
        return sequenceIndex;
    }
    else    //No active segment, this should not happen!
    {
        return -1;
    }
}
int HTM::getBestMatchingSegment(int c, int i, int t)
{
    //Find the segment with the largest number active synapses
    //The permanence value is allowed to be below connectedPerm
    //The number of active synapses is allowed to be below activationThreshold, but must↩
         be above minThreshold
    //If no segments are found, return -1

    int active_synapses, cell_index, col_index;
    int max_syn = minThreshold;
    int index = -1;

    for (unsigned s = 0; s < col[c].cell[i].segment.size(); s++)
    {
        active_synapses = 0;
        for (unsigned p = 0; p < col[c].cell[i].segment[s].potentialSynapses.size(); p↩
            ++)
        {
            cell_index = col[c].cell[i].segment[s].potentialSynapses[p].cell_index;
            col_index = col[c].cell[i].segment[s].potentialSynapses[p].column_index;
            if (col[col_index].cell[cell_index].activeState[t] == true)
            {
                active_synapses++;
            }
        }
        if (active_synapses > max_syn)
        {
            max_syn = active_synapses;
```

```
            col[c].cell[i].segment[s].num_active_synapses = max_syn;
            index = s;
        }
    }
    return index;
}
int HTM::getBestMatchingCell(unsigned c, unsigned t, unsigned wrongPredCell, int& seg)
{
    //Returns the cell with the best matching segment
    //If no matching cell: Return the cell with the fewest number of segments
    int s;
    seg = -1;
    unsigned num_active_synapses = 0;
    unsigned num_segments = col[c].cell[0].segment.size();
    int matching_cell_index = 0;
    int non_matching_cell_index = 0;
    for (unsigned i = 0; i < cellsPerColumn; i++)//int
    {
        s = getBestMatchingSegment(c,i,t);
        if (s !=-1)
        {
            if (col[c].cell[i].segment[s].num_active_synapses > num_active_synapses)    ↩
                //The number of active synapses must have been calculated here!!!
            {
                num_active_synapses = col[c].cell[i].segment[s].num_active_synapses;
                matching_cell_index = i;
                seg = s;
            }
        }
        if (col[c].cell[i].segment.size() < num_segments)
        {
            num_segments = col[c].cell[i].segment.size();
            non_matching_cell_index = i;
        }
    }
    if (num_active_synapses > 0)
    {
        return matching_cell_index;     //seg != -1
    }
    else
    {
        return non_matching_cell_index; //seg = -1
    }
}
HTM::segmentUpdate HTM::getSegmentActiveSynapses(int c, int i, int s, int t, bool ↩
    newSynapses)
{
    segmentUpdate seg;
    synapseUpdate synapse_tmp;
    std::vector<column> column_tmp = col;
    int cell_index, col_index, num;
    bool learn;

    seg.segIndex = -1;
    seg.sequenceSegment = false;

    if (s!=-1)
    {
        //Get active synapses
        seg.segIndex=s;
        seg.sequenceSegment=col[c].cell[i].segment[s].sequenceSegment;
        for (unsigned j = 0; j < col[c].cell[i].segment[s].potentialSynapses.size(); j↩
            ++)
        {
            if (col[c].cell[i].segment[s].potentialSynapses[j].permanence > ↩
                connectedPerm)
```

```
            {
                cell_index = col[c].cell[i].segment[s].potentialSynapses[j].cell_index;
                col_index = col[c].cell[i].segment[s].potentialSynapses[j].column_index;
                if (col[col_index].cell[cell_index].activeState[t] == true)
                {
                    synapse_tmp.cell_index = cell_index;
                    synapse_tmp.column_index = col_index;
                    synapse_tmp.synapse_index = j;
                    synapse_tmp.permanence = col[c].cell[i].segment[s].potentialSynapses↩
                        [j].permanence;
                    seg.activeSynapses.push_back(synapse_tmp);
                }
            }
        }
    }
    if (newSynapses)
    {
        num = max(0 ,int(maxSynapseCount - seg.activeSynapses.size())); //↩
            newSynapseCount in pseudo code = maxSynapseCount here
        while (num>0)
        {
            learn = false;
            for (unsigned c_i = 0; c_i < column_tmp.size(); c_i++)
            {
                for (unsigned i_i = 0; i_i < cellsPerColumn; i_i++)
                {
                    if ((i_i != i) || (c_i != c))   //No cell can connect to itself!
                    {
                        if (column_tmp[c_i].cell[i_i].learnState[t])
                        {
                            learn = true;
                            if (randInt(0,1))
                            {
                                column_tmp.erase(column_tmp.begin()+c_i);       //Do not↩
                                    look in this column again!
                                synapse_tmp.cell_index=i_i;
                                synapse_tmp.column_index=c_i;
                                synapse_tmp.permanence=initialPerm;
                                synapse_tmp.synapse_index = -1;
                                seg.activeSynapses.push_back(synapse_tmp);
                                num--;
                            }
                            break;
                        }
                    }
                }
            }
            if (!learn)
            {
                break;
            }
        }
    }
    return seg;
}
void HTM::adaptSegments(int c, int i, bool positiveReinforcement)
{
    //Adjusts permanences of synapses, and adds new synapses to the segment
    segment seg_tmp;
    syn syn_tmp;
    if (positiveReinforcement)
    {
        for (unsigned s = 0; s < col[c].cell[i].segment.size(); s++)
        {
            for (unsigned p = 0; p < col[c].cell[i].segment[s].potentialSynapses.size();↩
                p++)
```

```
            {
                col[c].cell[i].segment[s].potentialSynapses[p].permanence -= ↩
                    permanenceDec;
            }
        }
    }
    for (unsigned s = 0; s < segmentUpdateList[c][i].size(); s++)
    {
        if (segmentUpdateList[c][i][s].segIndex == -1 && segmentUpdateList[c][i][s].↩
            activeSynapses.size() > 0)
        {
            //New segment!
            seg_tmp.sequenceSegment = segmentUpdateList[c][i][s].sequenceSegment;
            for (unsigned a = 0; a < segmentUpdateList[c][i][s].activeSynapses.size(); a↩
                ++)
            {
                if (segmentUpdateList[c][i][s].activeSynapses[a].synapse_index == -1)
                {
                    syn_tmp.cell_index = segmentUpdateList[c][i][s].activeSynapses[a].↩
                        cell_index;
                    syn_tmp.column_index = segmentUpdateList[c][i][s].activeSynapses[a].↩
                        column_index;
                    syn_tmp.permanence = segmentUpdateList[c][i][s].activeSynapses[a].↩
                        permanence;        //Permanense = initialPerm!
                    seg_tmp.potentialSynapses.push_back(syn_tmp);
                    seg_tmp.num_active_synapses = 0;
                }
            }
            col[c].cell[i].segment.push_back(seg_tmp);
        }
        else if (segmentUpdateList[c][i][s].segIndex != -1)
        {
            //Old segment
            col[c].cell[i].segment[segmentUpdateList[c][i][s].segIndex].sequenceSegment ↩
                = segmentUpdateList[c][i][s].sequenceSegment;
            for (unsigned a = 0; a < segmentUpdateList[c][i][s].activeSynapses.size(); a↩
                ++)
            {
                if (segmentUpdateList[c][i][s].activeSynapses[a].synapse_index == -1)
                {
                    syn_tmp.cell_index = segmentUpdateList[c][i][s].activeSynapses[a].↩
                        cell_index;
                    syn_tmp.column_index = segmentUpdateList[c][i][s].activeSynapses[a].↩
                        column_index;
                    syn_tmp.permanence = segmentUpdateList[c][i][s].activeSynapses[a].↩
                        permanence;        //Permanense = initialPerm!
                    col[c].cell[i].segment[segmentUpdateList[c][i][s].segIndex].↩
                        potentialSynapses.push_back(syn_tmp);
                }
                else
                {
                    if (positiveReinforcement)      //Positive Reinforcement
                    {
                        col[c].cell[i].segment[segmentUpdateList[c][i][s].segIndex].↩
                            potentialSynapses[segmentUpdateList[c][i][s].activeSynapses[↩
                            a].synapse_index].permanence += (2*permanenceInc);
                    }
                    else        //Negative Reinforcement
                    {
                        col[c].cell[i].segment[segmentUpdateList[c][i][s].segIndex].↩
                            potentialSynapses[segmentUpdateList[c][i][s].activeSynapses[↩
                            a].synapse_index].permanence -= permanenceDec;
                    }
                }
            }
        }
```

```
    }
}
/*Helper functions for Temporal Pooler*/
void HTM::initTimeStep()
{
    /*Add entry for current time step*/
    for (unsigned c=0; c < col.size(); c++)
    {
        for (unsigned i = 0; i < cellsPerColumn; i++)
        {
            col[c].cell[i].activeState.push_back(false);
            col[c].cell[i].predictiveState.push_back(false);
            col[c].cell[i].learnState.push_back(false);
        }
        col[c].surpriseState.push_back(false);
    }
}
float HTM::getSegmentActivity(int c, int i, int s, int t, state cellState)
{
    //Moving average not yet implemented here, just return the first segment
    if(segmentActive(c,i,s,t,cellState))
    {

    }
    return 1;
}
void HTM::updateCellSynapses(int c, int i)
{
    //Update connected synapses!
    bool noSynFlag;
    for (unsigned s = 0; s < col[c].cell[i].segment.size(); s++)
    {
        noSynFlag = true;
        col[c].cell[i].segment[s].connectedSynapses.clear();
        for (unsigned p = 0; p < col[c].cell[i].segment[s].potentialSynapses.size(); p↩
            ++)
        {
            if (col[c].cell[i].segment[s].potentialSynapses[p].permanence > ↩
                connectedPerm)
            {
                col[c].cell[i].segment[s].connectedSynapses.push_back(col[c].cell[i].↩
                    segment[s].potentialSynapses[p]);
                noSynFlag = false;
            }
        }
        if(noSynFlag)
        {
            col[c].cell[i].segment.erase(col[c].cell[i].segment.begin()+s);
        }
    }
}
```

# B Qt application output

The figure below shows a test with the input bit sequence 010101. There are six sub figures (a,b,c,d,e,f), one for each bit in the input bit sequence. The HTM consists of one layer that has two column with three cells each. When the input bit is zero the left column is activated, and when the input bit is one, the right column is activated. A blue cell denotes a cell in the active state, and a yellow cell denotes a cell in the predictive state. If a cell is gray it is has no active input and is neither active nor predictive. From the beginning there are no connections between cells, but as the learncell in a winning column is forming connections to recently activated cells, it will soon start to predict its own activation. By doing this the cells in the columns will learn the input bit sequence. Connections between cells are shown by thin black lines in the figure.

**(a)** *The input is zero, and the left column is surprised. It choses a learncell but cannot create any connections.*

**(b)** *The input is one, the right column is surprised. It choses a learncell that creates a connection to a cell that was chosen as learncell in the last time step. (in this case to the first cell in the left column).*

**(c)** *The input is zero, and the left column is surprised. It choses a learncell and creates a connection to a cell that was chosen as learncell in the last time step. The first cell in the right column receives active lateral input and enters the predictive state.*

**(d)** *The input is one, and the right column is now no longer surprised because it had a cell in the predictive state in the last time step. Only that cell is now active, and no new connections are created. The first cell in the left column is now in predictive state and anticipates activation the next time step.*

**(e)** *The input is zero, and the first cell in the left column is in the active state as predicted. The first cell in the right column is in the predictive state and anticipates activation the next time step.*

**(f)** *The input is one, and the first cell in the right column is in the active state as predicted. The first cell in the left column is in the predictive state and anticipates activation the next time step.*

**Figure B.1:** *The output states of the cells in two columns during a simple input sequence.*