1. Introduction

1.1. Motivation

HTM (Hierarchical Temporal Memory) is an unsupervised on-line machine learning system initially developed by Numenta. Although M-HTM (Multilayer Hierarchical Temporal Memory) were already proposed in Numenta's early papers, actual reference implementations do not exist yet. Several papers explore M-HTM setups, none of which implement Temporal Pooling as described by Numenta. According to Numenta, M-HTMs in combination with Temporal Pooling should be able to solve complex problems, such as image recognition[1].
In this project, a new approach to M-HTMs including Temporal Pooling is explored and evaluated using a practical problem rooted in robotics.

1.2. Project Description

This project extends a previously implemented HTM by creating a layered hierarchy of multiple HTM regions. The C-implementation is parallelized using multiprocessing on the level of regions and multithreading on the level of columns. The software is tested and evaluated in several steps. First, simple randomly generated integer patterns are given as input for both single and multihierarchical region setups. Afterwards, coordinate and sound data produced by a robot with speakers are generated and used on an M-HTM. The performance of these setups was to be evaluated.

1.3. Background

This project is based on the two bachelor theses "Embedded Cortical Learning with Hierarchical Temporal Memory"[2] and "Implementation of a Cortical Memory Model on the SpiNNaker Neuromorphic Hardware"[3]. The program of the latter is used as a basis and extended upon, because the implementation of the first one is heavily specialized for its architecture. Having read any of the two theses or Numenta's whitepapers[1] is helpful to understand this report.

1.4. Technical Specifications

The M-HTM implementation was tested on Ubuntu 16.04 Xenial Xerus but should run on any modern Unix System. The implementation was realized in C with gcc version 5.4. The sound and coordinate data was generated with a modified Version of the Robotino2.

3. Multilayer Hierarchical Temporal Memory

3.1. Temporal Pooling

The algorithms as currently described by Numenta use Temporal Memory to learn the transitions between tuples of input values [1]. This allows the HTM to predict one single timestep into the future. But Numenta's papers also mention Temporal Pooling which is an idea on how to predict multiple timesteps into the future [1]. By doing so, an abstract representation of the whole pattern can be formed and predicted as soon as the HTM recognizes enough input values to identify the whole pattern [1]. Temporal Pooling is particularly interesting for M-HTMs since the abstract representations can be communicated upwards in the hierarchy and thereby achieving temporal abstraction between the layers of the M-HTM.

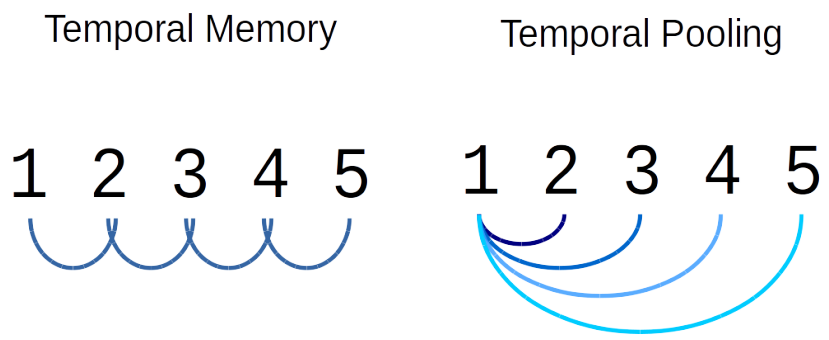

Figure 1 Illustration of difference between Temporal Memory and Temporal Pooling

The desired behavior of Temporal Pooling was achieved by modifying the code corresponding to Temporal Memory. The idea behind the modifications is as follows: By increasing the number of cycles a cell can stay in the predictive/learning state, the cell can learn to predict its activation multiple timesteps into the future. This number is fixed for each cell and randomly set at initialization time. This leads to multiple different timeframes the HTM can predict simultaneously instead of only one timestep. After the trained HTM is given a few values of a pattern, it can predict the whole pattern while the output of the HTM (the set of predictive cells) barely changes as long as the pattern does not change. The output can then be treated as the abstract representation of the pattern.

3.2. Predictive Overlap

The predictive overlap is defined as the percentage of inherited predictive columns of the previous cycle. This will be referenced as overlap from this point on and is not to be mistaken with the overlap function in Spatial Pooling. An example can be seen in figure 2. Based on temporal pooling, a pattern should have a stable representation which can be given to a higher level region. This affects the overlap such that it has a high overlap within the pattern and low overlap outside the pattern or

when random values appear. This means a few values into a pattern, the stable representation is achieved with a high overlap. Upon a pattern change, the overlap immediately drops and quickly builds back up when progressing into the new pattern. The edge case of very short patterns leads to a problem. In our tests it was found that a pattern of length three or lower would not form an abstract representation.

cycle x-1	0 1 0 0 0 1	0 1 0 0 0 1
cycle x	0 1 0 1 0 1	0 1 0 1 1 0
Overlap	100%	50%

Figure 2 Example for the calculation of the predictive overlap

3.3. Sparse Distributed Input for High Level Regions

Since low level regions receive the sparsified input of test or generated data, the regions above get their sparse input from the lower regions. The higher region is expected to build predictions upon previous predictions. It follows that the input a high level region receives is the predictive state of a lower level region. In particular, this means that the lower region sends a string of bits with one bit for each of its cells to the upper region. If a cell is predicitive the respective bit is set to one otherwise it is zero. By the nature of the HTM, the resulting string should keep its sparsity. If that is not the case various parameters can be adjusted to reduce the number of set bits. When multiple regions feed their predictive states to the same region the outputs of the lower regions are simply concatenated. While progressing in a pattern its abstract representation does not change significantly. Transmitting the same representation repeatedly to the higher layer is counterproductive since the higher region should learn a pattern of abstract representations of lower level patterns. Therefore, an overlap threshold is introduced. The corresponding pseudocode can be seen in table 2. The lower region should ideally transmit once per pattern using the method described in the pseudocode. This essentialy negates repeated transmitting and also transmission of single anomalies since these do not produce overlap.

```
1 boolean flag = false;
2 for(int cycle = 0; cycle != finish; cycle++){
3   if(overlap > threshold && !flag){
4     transmit_predictive_states();
5     flag = true;
6   }
7   else if(overlap < threshold)
8     flag = false;
9 }
```

Table 2 Pseudocode structure for the transmission of predictive states by lower level regions

However, this approach does have its caveats when using multiple regions feeding to a single one. Since the higher level region requires a concatenation of the lower regions output all lower regions should transmit at the same rate, meaning every x cycles with x being the pattern length of the lower inputs. This would imply that all lower regions have the same learning performance which is improbable. This leads to a desynchronization between the regions in terms of their transmission to the higher region. None the less, the higher regions is able to adjust and learn the desynchronization in terms of pattern if the difference is stable. For example if region 1 transmits about twice as often as region 2, then the upper region will simply learn a repeating pattern that is twice as long as it would normally be. Although not optimal, this is still significantly better then transmitting after every single input of the lower regions. In practice pattern length will not be stable therefore increasing the impact of the desynchronization problem. A suitable solution has to be found in order to make M-HTM feasible for practical applications. A possibility might be to make every lower region transmit their predictive state, if any of the regions would. This approach would get rid of desynchronization, but would introduce some repetitive transmissions.

4. Implementation

This chapter is a collection of new functionalities and code segments with respective explanations excluding the multihierarchical changes that were explained in detail in chapter 3.

4.0.1. Global Configuration

While the region hierarchy can also be seen as part of the global configuration, the file was split off for various design considerations. Currently, the *global_configuration.txt* only contains the two parameters *number_regions* and *cycles*. *number_regions* simply declares the number of regions in this M-HTM whereas *cycles* gives the number of running cycles of regions without incoming communication edges. Regions of higher levels do not neccessarily run as many cycles as lower ones in the current implementation, because higher regions only receive input when the overlap of lower regions change.

4.1. Config Files

4.1.1. Region Hierarchy

The *region_hierarchy.txt* file sets the configuration for the communication of regions. The file starts with the actual number of regions and is followed by a number of tuples each in their respective line. This can be read as directed edges in a graph. For example the tuple (1, 0) would indicate a communication from region with id 1 to region with id 0. In terms of layers, this would put region 0 above region 1.

Region hierarchies resulting in circles (i.e. feedback loops) do not terminate in this version. The circle results in a circular waiting system and produces a deadlock. This could be solved by declaring some region as the "first" region and simply let the chosen region receive a null input. That way output can be produced and the deadlock is lifted. This "first" region would need to be defined for every cycle in the graph. This would also solve the issue with regions connecting to themselves as those are simply circles of length one.

4.1.2. Region Configuration

Every Region uses a different configuration file named *region_id_a.txt* with a being some integer. This file contains a multitude of parameters. The description for these parameters can be seen in table 4.1.2. For every region, a configuration file is required to run the program. Please note, that the region IDs need to start at zero and increment up to the number of regions minus one.

Table 3 List of configuration parameters of the M-HTM implementation

COLUMN_COUNT	Number of columns in the region.
CELL_COUNT	Number of cells per column.
INPUT_COUNT	Number of input connections per column to the SDR.
INPUT_PERMANENCE_THRESHOLD	Threshold of permanence at which the input connection is

	considered active.
INPUT_PERMANENCE_INC	Value at which the permanence of the input connection is increased when reinforced.
INPUT_PERMANENCE_DEC	Value at which the permanence of the input connection is decreased when reinforced.
INPUT_PERMANENCE_CHECK	Boolean value whether the permanence value of the input connection is checked before activation.
COLUMN_STIMULUS_THRESHOLD	Value of overlap which must be exceeded for the column to be considered a candidate for activation.
COLUMN_MAX_BOOST	Maximum boost value for all columns.
COLUMN_START_BOOST	Cycle at which boosting starts.
COLUMN_AVERAGE_WINDOW	Length of the average window for boost approximation.
REGION_ACTIVE_COLUMNS	Columns that are in the top X of overlap values will be activated with X being said parameter.
CELL_REMAIN_ACTIVE	Maximum number of cycles a cell will remain active.
CELL_REMAIN_PREDICTIVE	Maximum number of cycles a cell will remain predictive.
CELL_REMAIN_LEARNING	Maximum number of cycles a cell will remain learning.
CELL_REMAIN_RANDOM	Boolean value whether the remain value should be fixed or randomized uniformly up to their respective maximum per cell.
SEGMENT_ACTIVATION_THRESHOLD	Number of connected active connections a segment must exceed to become active.
SEGMENT_LEARNING_THRESHOLD	Number of connected active connections a segment must exceed to become learning.
SEGMENT_NEW_CONNECTIONS	Maximum number of new connections added to a segment per cycle.
CONNECTION_LEARNING_HORIZONTAL	Maximum horizontal distance for new connections. The topology is a one dimensional line. Column 0 has distance 100 to column 100.
CONNECTION_LEARNING_VERTICAL	Maximum vertical distance for new connection. Every column contains multiple cells. This allows for cells to learn from multiple cells of a column or even from their own column.
CONNECTION_PERMANENCE _THRESHOLD	Threshold of permanence at which an internal connection is considered active.
CONNECTION_INITIAL_PERMANENCE	The initial permanence of any given internal connection.
CONNECTION_PERMANENCE_INC	Value at which the permanence of the internal connection is increased when reinforced.
CONNECTION_PERMANENCE_DEC	Value at which the permanence of the internal connection is decreased when reinforced.
FORGET_INTERVAL	The interval at which internal connections or segments are removed if unused for this amount of cycles.
DETECTION_THRESHOLD	Detection threshold for anomaly detection. Example: $D_T = 0.6$.

	If more than 60% of active columns burst, the input is considered an anomaly.
OVERLAP_THRESHOLD	Overlap threshold at which this regions output is given to a higher region. A more detailed explanation can be found in section 3.3.
ENABLE_LEARNING	Boolean value whether learning should be enabled or not.
LOAD	Boolean value whether this region should load a previous state. The corresponding file is required.
SAVE	Boolean value whether this region should save its state after the full termination. The state will be written to a file.
var	Artificial Variance number for input noise. Primarily used for testing.
random_prob	Probability of a given value in a pattern to be replaced with another value. Primarily used for testing.
warmup	Number of cycles the region learns before the statistics start being measured.
cooldown	Number of cycles the region cools down after an anomaly before the next anomaly can be detected.

4.2. Saving and Loading

The implementation supports loading and saving the current state of a region. Both saving and loading can be switched on or off individually for any region. The data is saved in a text format and encompasses all neccessary region information. A region always loads the save file with the same ID. Thus, *region 1* would load *region_id_1.dat*. Saving overwrites the old save file of the same ID.

4.3. Scripts

There are currently two scripts available. The *build.sh* simply compiles the code and takes no parameters. The *run.sh* takes the number of regions in the M-HTM as input parameter and starts all regions. The region executions are saved as logfiles in the *logs* directory. If any duplicate logfile would be created, the script instead exits with a warning. Thus, logfiles are never overwritten.

4.4. Parallelization

4.4.1. Multi-processing with Named Pipes

In order to enable communication between the processes each running one region, named pipes are used. If the Network of HTMs is considered a graph then every directed edge corresponds to one named pipe between those two regions. In every cycle two data segments are transferred. The

first is the cycle number. As described in section 3.3 the lower level regions are not neccessarily in the same cycle. The difference between multiple cycle numbers of lower regions represent the desynchronization between them. The second segment is the predictive state after the corresponding cycle and is used to transmit the input to the higher region. The named pipes are maintained by every region as two lists of read and write pipes. If the list equals *NULL* then no corresponding pipe exists for this region.

4.4.2. Multi-threading with pThreads

Parallelization inside regions was realized by implementing the thread-pool design pattern. This implementation of the thread-pool is a C-struct which manages a fixed number of pThreads. Jobs can be submitted to the thread-pool which will then be executed in parallel. HTM-functions that represent logic which can be executed in parallel are packaged into jobs and submitted to the thread-pool.

4.5. Disabling Learning

Learning of any one region can be disabled with the corresponding parameter in the region configuration. In particular, the addition as well as the reinforcement of segments and connections is disabled. This includes the boosting value. When used with saving and loading the program can now train a region once and then reuse that region without changing it by disabling learning.

5. Methodology

5.1. Definitions

Inputs are classified into two subcategories. An input can belong to a *pattern* or be a *random value*. The HTM tries to classify the input as one of the two categories. An input is *correctly identified*, if the HTM classification matches the actual classification of the input. This classification is based on the ratio of *bursting columns* to *active columns*. If this ratio is below a *detection threshold*, an input is classified as a pattern. Otherwise it is classified as an anomaly. The classification accuracy of either is called *Pattern Recognition Rate* and *Anomaly Detection Rate*. They are formally defined as follows. Note that every cycle gives one input value.

Definition 1. *Let C_i denote the number of cycles in trial* i *and R_i the number of correctly identified pattern inputs in trial* i *with* n *being the number of trials. Then the* **Pattern Recognition Rate r** *is defined as:*

$$\frac{\sum_{i=0}^{n} C_i/R_i}{n} = \sum_{i=0}^{n} \frac{C_i}{R_i * n} \tag{5.1}$$

Definition 2. *Let C_i denote the number of cycles in trial* i *and D_i the number of correctly identified anomaly inputs in trial* i *with* n *being the number of trials. Then the* **Anomaly Detection Rate d** *is defined as:*

$$\frac{\sum_{i=0}^{n} C_i/D_i}{n} = \sum_{i=0}^{n} \frac{C_i}{D_i * n} \tag{5.2}$$

5.2. Sound Data Collection

For the live collection of movement and sound data several hardware pieces were required. For movement, a Robotino2 was used. A speaker was mounted on the Robotino. The Robotino would then move in straight paths while playing some sound with a microphone positioned in the room. An illustration of the setup can be seen in Figure 3.

First, a sine wave of 620hz was used as the sound sample. However, this lead to the reflexion of sound waves cancelling out at certain positions in the room depending on the speaker position. Since the only important information from the sound in this setup is the amplitude on the left and right audio channel, the audio simply needed to be loud enough and to some degree constant.
After a multitude of sound sample tests, some rain soundtrack was used as a sample. The random noise in the rain sample would introduce some level of noise to the data which is important for evaluating the noise robustness of our M-HTM. The recording was done at 44.1khz resulting in $44,100$ samples per second. Because our computer hardware can not handle HTMs designed for such a pattern length, the number of samples was reduced. For this, the sample were split into subsections of samples. For example, when aiming for 500 samples in 10 seconds of soundtrack, the $441,000$ samples were grouped into sets of $22,050$ samples which is $1/500$-th of the initial value. For every

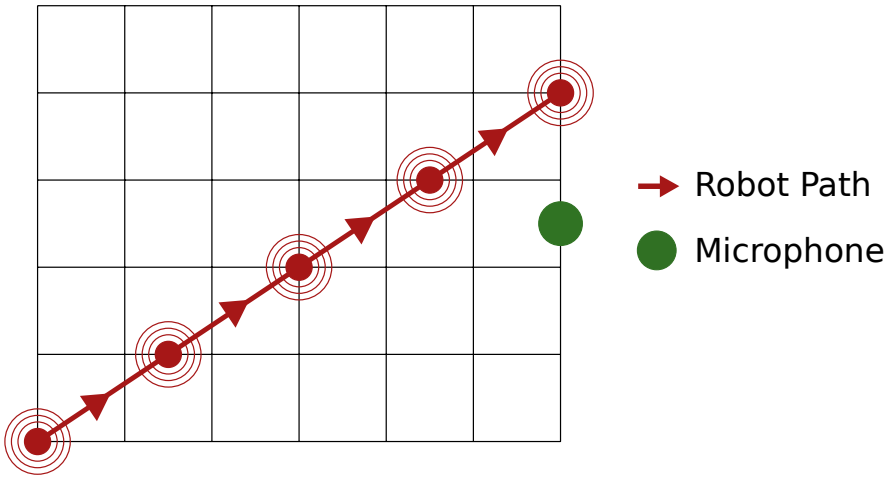

Figure 3 Simplified depiction of the sound data collection process

group, the peak amplitude value was extracted to generate the final pattern.
The goal is to correlate the sound to the coordinate data. The peak amplitude values should steadily increase, decrease or shift from left to right channel depending on the movement of the robotino/speaker in the room.

6. Evaluation

6.1. Single Region Performance Test

6.1.1. Setup for Single Region Test

The randomly generated inputs consisting of patterns and random values (anomalies) are used to test the theoretical capabilities of a single HTM region. Every test case consists of n valid trials where n is specified for every test case. Every trial of a given test case uses the same parameters. The two calculated data points are *Pattern Recognition Rate* and *Anomaly Detection Rate*.

A pattern A of length $|A| = n$ consists of multiple uniformly distributed random integers $a_i, 0 < i < n-1, 0 < a_i < SDR_b$ and is fixed for every test case. SDR_b describes the SDR Base length and is defined for every test case. The individual values a_i are fed to the HTM region beginning at a_0 up to $a_n - 1$. In order to introduce anomalies, with a chance of $1/(2n)$, a pattern input is exchanged with a random number of the same range as a_i.

A test case uses x different randomly generated patterns. After a pattern is fully fed to a region, with a chance of $1/(2x)$, a set of n random values is given to the region. Otherwise, a new pattern is randomly chosen to be the next pattern that will be given to the region.

Every trial has a warm up phase of c_w cycles to learn the given patterns before results are collected. Note that anomalies are still applied in this phase. In addition, every trial runs for c cycles. Both c and c_w are persistent throughout a test case.

Note that by defining the number of anomalies introduced this way, the percentage of actual anomalies is reduced with a higher number of patterns. This was done for multiple reason. The first one being a simply practical. The computer that was used for the evaluation was running trials for more than a week. This time includes parameter optimization. In addition, it was found that increasing the number of anomalies beyond this barely affects the detection rates. In singled out experiments, the percentages were found to be a few percent lower, using suboptimal parameter settings.

6.1.2. Results of the Single Region Test

In order to evaluate the performance of a single region, six test cases were performed. Each of these test cases consist of 50 trials. For all test cases the temporal pooling values *REMAIN_PREDICTIVE* and *REMAIN_ACTIVE* were equal to the pattern length. The remaining specifications can be seen in Table 4. This table only contains values that changed between the executions. All other parameters are on their base setting in the code submission.

The performance results of the respective test cases can be seen in Figure 4. Note that these results could further be improved by using larger and different region parameters. This would result in an increase of the runtime. The tuning of variables presented here was done so that the runtime was feasible for the attained result. Test case 6 was the longest run, taking about 23 minutes per trial resulting in approximately 11 hours and 10 minutes on an Intel i7 CPU. This is just an indicator of the runtime on a modern CPU and will vastly differ on different architectures.

Test cases 2 through 4 represent a baseline. These test cases only increase in the number of

Table 4 Single Region Test: Specifications

Case 0		**Case 1**		**Case 2**	
Number pattern:	100	Number pattern:	20	Number pattern:	1
Pattern length:	10	Pattern length:	30	Pattern length:	10
Cycles:	9000	Cycles:	12000	Cycles:	4000
Warmup:	4000	Warmup:	5000	Warmup:	1000
Columns:	3000	Columns:	3000	Columns:	2000
Cells per column:	30	Cells per column:	30	Cells per column:	20
Detection threshold:	0.9	Detection threshold:	0.9	Detection threshold:	0.5
Random value:	1/20	Random value:	1/60	Random value:	1/100
Random pattern:	1/200	Random pattern:	1/40	Random pattern:	1/2
Case 3		**Case 4**		**Case 5**	
Number pattern:	5	Number pattern:	10	Number pattern:	5
Pattern length:	50	Pattern length:	50	Pattern length:	200
Cycles:	5000	Cycles:	8000	Cycles:	14000
Warmup:	2000	Warmup:	3000	Warmup:	7000
Columns:	2000	Columns:	2000	Columns:	3000
Cells per column:	20	Cells per column:	20	Cells per column:	30
Detection threshold:	0.4	Detection threshold:	0.5	Detection threshold:	0.9
Random value:	1/100	Random value:	1/100	Random value:	1/400
Random pattern:	1/10	Random pattern:	1/20	Random pattern:	1/10

patterns with a pattern length of 50. In contrast, test cases 0 and 5 are extreme cases with a higher number of patterns and lower pattern length and vice versa. Test case 1 is an example of medium pattern length with medium number of patterns. It can be seen that both extreme cases have a comparably low pattern recognition with about 86-87%. Since the test case 0 with a high number of patterns only has a pattern length of 10, 90% or less pattern recognition rate is to be expected as the first value of every pattern is unexpected. This shows that the general performance worsens with longer patterns, but not as much with the number of patterns.

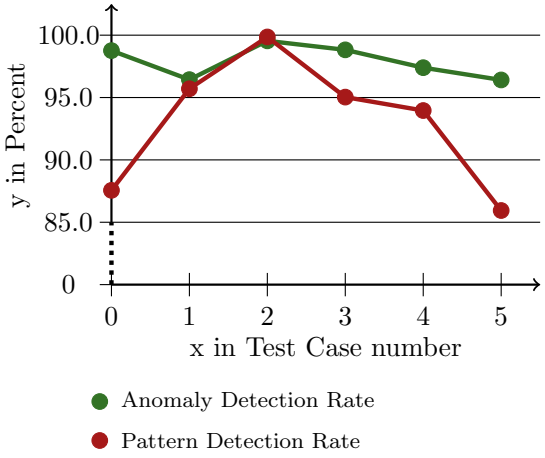

Figure 4 Single Region Performance: Test cases 0 – 5

6.2. Two-Layer Abstraction Test

This two-layer setup shares most of the setup in terms of definitions with the single region setup. The test uses 25 patterns of length 10 to train the lower regions. The SDR base length for the test is 1000. 50 trials are run for the evaluation of the two-layer abstraction test. The evaluated setup uses two regions with one being connected to the other in a one to one setup.
However, testing the higher level region is more complex. In total, the lower region ran for 12000 cycles. The first 4000 cycles were used for training the lower region on the 25 different patterns. In cycles 4001–8000 the upper region was trained on the abstract representations given by the lower region while learning was disabled in the lower region from cycle 4001 onward. From cycle 8001–12000 the actual test took place since in this phase in every 500th cycle, 10 random values instead of a pattern were fed to the lower region, causing a time shift. The upper region was expected to detect said anomaly successfully (*anomaly detection rate*). If the upper region registered at least one anomaly up to 15 cycles after the first random value or i.e. up to 5 cycles after the last random value, the anomaly was detected successfully.
False anomalies were also collected (*false positive rate*), meaning the upper region registered an anomaly when there was none (i.e. outside said window). The last data point that was collected is the ratio of upper region cycles to lower region cycles (*cycle ratio*). The optimal ratio is 1 to 10, as this implies one abstract representation being communicated per pattern of length 10.
The results can be seen in Table 5. The results show that the setup is capable of detecting most anomalies and that the cycle ratio is close to the optimal 10. An explanation as to why it is below and not above ten is that the lower region learns some of the transitions between the different patterns. This results in the lower region transmitting on average slightly less than once per pattern.
For a full evaluation on multiple lower regions connected to a higher region, the scope of this project ended up being too small. When running similar tests as above the upper region was capable of detecting anomalies but was weaker in terms of pattern recognition. This is due to the desynchronization between the lower level regions.

Table 5 Two-Layer Abstraction Test: Setup and results

Pattern count:	25	Pattern length:	10
Cycle count:	12000	Column count:	1000
Cells per column:	10	Anomaly detection rate:	0.8925
False positive rate:	0.0185	Cycle ratio:	0.0959

6.3. Negative Result with Sound Data Learning

When using sound data as described in the methodology, the HTM is incapable of differentiating the sound tracks after training the HTM on multiple tracks. To obtain this result, six patterns of straight (0°relative to x-axis) paths beginning at different positions of the coordinate grid were learned. Then

learning was disabled and a seventh pattern was given to the HTM. This pattern was unknown to the HTM and should not have been recognized. However, the HTM reported a recognition rate of 82%. It follows that the generated sound data is too similar even when reducing the spatial overlap of the base SDRs.

A correlation between the sound data and any other data would have been impossible this way. The reason for this is the employed preprocessing of the sound data and not the HTM itself. A better way of preprocessing might yield better results. However, adopting a more complex approach was not possible due to a lack of remaining time.

7. Conclusion and Future Work

The existing code base was extended to include support for parallelization on general purpose computer architectures. Also, methods for linking multiple HTM regions to a hierarchical M-HTM setup were developed. To achieve temporal abstraction of patterns between the layers of a M-HTM setup, Temporal Memory was modified to a prototypical Temporal Pooling algorithm. Said algorithm showed promising results when evaluating a simple two-layer M-HTM on generated patterns. It achieved temporal abstraction without compromising pattern recognition and anomaly detection on any layer. To determine the feasibility of M-HTM for practical applications in robotics, further research is required. The desynchronization of multiple lower level regions compromises pattern recognition if said hierarchy is set up. A solution for this problem is required to make larger M-HTM hierarchies viable. Another interesting area of inquiry is the application of M-HTM in the context of cluster computing. The current implementation could be extended to support OpenMP and MPI with little effort.

List of Figures

List of Tables