Towards a Neuromorphic Implementation of Hierarchical Temporal Memory on SpiNNaker

Florian Walter, Marwin Sandner, Florian Röhrbein, Alois Knoll
Department of Informatics, Chair of Robotics and Embedded Systems
Technical University of Munich, Garching bei München, Germany
Email: florian.walter@tum.de, marwin.sandner@tum.de, florian.roehrbein@in.tum.de, knoll@in.tum.de

***Abstract*—Hierarchical Temporal Memory (HTM) is a computational model of the neocortex that is capable of online learning to predict and detect anomalies from continuous data streams. To make HTM also available on power-constrained robot systems, we investigate the feasibility of implementing the model on SpiNNaker, a fully programmable energy-efficient neuromorphic many core system. Our contribution is twofold: First, we propose a mapping of the HTM model components to the SpiNNaker chip architecture. Second, a prototypic implementation of this mapping is successfully evaluated for different sets of model parameters.**

I. Introduction

Hierarchical Temporal Memory (HTM) is an abstract neural model that is claimed to capture functional principles of neocortical computation [1], [2]. It is capable of learning online from continuous data streams to predict future input and to detect anomalies. This makes the HTM model very attractive for robotics where it could be used to predict future sensor input for prospective control and to automatically detect malfunction and failure. However, the low processing power of embedded control units and the high power consumption of modern CPUs and GPUs currently preclude running the model at a practically relevant scale on mobile robot systems.

The poor performance of standard von Neumann microprocessors at simulating neural networks motivated the development of neuromorphic chip designs. Most current architectures are specifically tailored to the simulation of spiking neuron models and biologically plausible synaptic plasticity rules [3]. Accordingly, they outperform standard processors considerably in terms of both simulation speed and power efficiency. This makes it possible to set up physical closed-loop neurorobotic systems [4] in which robots are directly connected to real-time simulations of large-scale brain models such as those currently developed in the European Human Brain Project [5], [6].

In spite of the general focus on spiking neural networks, not all neuromorphic chip designs are limited to this class of neural networks. The SpiNNaker architecture [7] is based on standard ARM cores that can execute arbitrary user-defined code, which makes it in principle also capable of simulating more abstract brain models like HTM. Besides its power efficiency, another special advantage of SpiNNaker over other architectures is its flexible on-chip routing that scales from systems with only a few cores to clusters with thousands of cores. Moreover, SpiNNaker is already widely used in neurorobotics. Successful applications include the control of a biomimetic robot arm based on a spiking neural model of the cerebellum [8] and a mobile neuromorphic robot platform [9].

Motivated by the successful case studies of using SpiNNaker in neurorobotics, this paper reports on our work on porting the HTM model to the SpiNNaker architecture. Differently from a previous study that focused on converting single HTM components to a spiking neuron representation [10], our goal is to deliver an initial prototypic implementation of the HTM algorithms that runs natively on the SpiNNaker cores. The remainder of the paper is structured as follows: In the next section, we provide a brief introduction to SpiNNaker and HTM. In section III, we propose a mapping of the HTM model components to a single SpiNNaker chip and briefly describe our implementation of that mapping. The results of an initial evaluation are summarized in section IV. Section V finally concludes the paper and outlines the next steps that are required to apply our initial prototype on a robot.

II. Background: SpiNNaker and HTM

The following two paragraphs summarize the most important aspects of the SpiNNaker architecture and the HTM model. For more detailed descriptions, the reader is referred to [7] and [1], [2], respectively.

SpiNNaker is a digital neuromorphic many core system. Each chip is comprised of 18 ARM968 cores that run at 200 MHz, 128 MB shared memory, and on-chip router that is optimized for packet communication based on the address event representation protocol. All chips are interconnected in a mesh with toroidal topology that can scale up to one million processor cores. The cores exchange information via the SpiNNaker Datagram Protocol (SDP). Communication is performed asynchronously. In particular, there are no guarantees regarding packet ordering and packets may be even dropped. The system is programmed in the C programming language and only supports fixed point arithmetic. SpiNNaker applications follow an event-driven programming scheme and run without an operating system. Instead, every SpiNNaker application is linked to the SpiNNaker Application Runtime Kernel (SARK).

HTM is a machine learning model that is inspired by the neocortex. The basic building blocks of the HTM model are cells that can assume different discrete states. Cells are grouped in columns of identical size. A set of columns forms a region. All cells in a column receive binary input via a shared proximal dendrite segment that is connected to a subset of the input bits. A spatial pooler activates only those columns in a region which receive sufficient input. On activation of a column, active connections are positively reinforced. Similarly, the temporal memory learns temporal patterns in the input data

978-1-4673-6853-7/17/$31.00 ©2017 IEEE

by reinforcing distal dendrite segments that connect a cell of one column to cells in other columns in the region. Together, these two mechanisms enable learning from continuous data streams for prediction and anomaly detection. All input data must be encoded in a sparse distributed representation.

III. Implementation

In the next subsections, we will discuss our implementation of the HTM model as defined in [1] and [2] for the SpiNNaker system. The focus is on an appropriate parallelization of the HTM learning algorithms, which is a prerequisite for an effective usage of SpiNNaker's computational resources. At many points, our implementation is different from the HTM reference code by Numenta which is designed for standard general purpose computers and therefore not compatible with the SpiNNaker platform. Throughout the whole text, we consider mapping one HTM region one SpiNNaker chip. In the future, this scheme can be easily extended to running a network of multiple regions on a SpiNNaker board and thereby making full use of the modular hardware architecture. Our HTM model for SpiNNaker is implemented in the C programming language. While the core of the machine learning system follows a structured and imperative programming paradigm, the code used for parallelizing the system for the SpiNNaker architecture is event-driven. Whenever possible, we directly adopt the algorithms described in [2].

A. Architecture

The data structures which represent the different components of a HTM region are implemented as simple C structures that are composed in a tree-like hierarchy. Operations on HTM regions are implemented as functions that iterate over the set of all columns and manipulate them as well as their cells and dendrite segments individually. Many of them only depend on the current state of the column that is modified or on state variables of other columns from previous time steps. Consequently, the different HTM operations can be parallelized at the granularity level of single columns. The layout of an instantiated HTM model on the SpiNNaker board is depicted in Fig. 1: One region is allocated to one chip. The data structures reside in the shared SDRAM of the respective chip. This is the only memory on a SpiNNaker chip that offers enough space and is accessible by all of the chip's

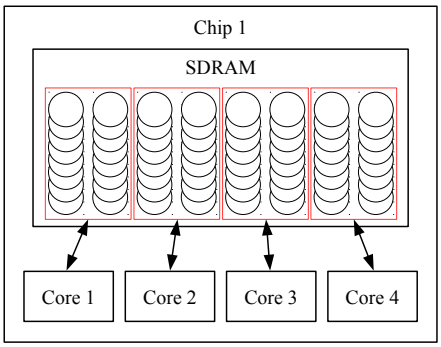

Fig. 1. Mapping of the HTM model to a SpiNNaker chip. In this example, the HTM region is comprised of eight columns (with eight cells each) that are mapped to four of the chip's cores (red boxes). In practice, all cores of a chip are occupied by one region (e.g. 500 columns ↔ 17 cores).

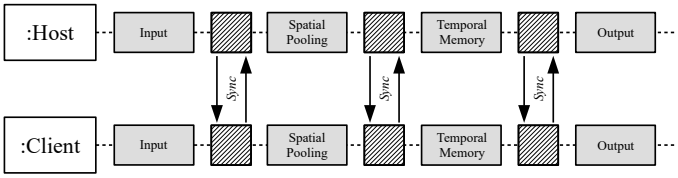

Fig. 2. Simplified flowchart of the communication between the host and one client during an HTM update cycle. The squares with diagonal lines indicate synchronization barriers. In practice, parallelization is more fine-grained with multiple barriers in each of the depicted phases. The barriers are realized with a simple handshake protocol.

cores [7]. The possibility of designing the architecture such that one region can be allocated and executed on multiple chips was discarded in this initial study to avoid considerable synchronization overhead.

B. Parallelization

As depicted in Fig. 1, each core is assigned an equally large set of columns. One of the cores in a chip is chosen as the host, the other cores become clients. All of the functions that meet the conditions described earlier are executed in parallel by every core. They manipulate only the columns assigned to the corresponding core, alleviating the need for any locks or semaphores since no data races can occur by design. All other functions are executed by the host while the clients wait for completion. The synchronization between the particular phases of the spatial pooling and the temporal memory is managed by the host. It signals the clients at the beginning of a new phase and then, after having completed the computations of that phase locally, also requests their completion status. When all clients have confirmed completion, the host starts the next phase by signalling all clients. A simplified schema of this handshake protocol is depicted in Fig. 2. The cores communicate via SDP messages. All messages are allocated by the SpiNNaker API and each of them is assigned an ID to convey the context of the message (i.e. the current phase in the cycle). A message may also contain pointer addresses to structures residing in the shared memory such as the pointer to the region structure. Because of the unreliable nature of the protocol [11], the clients are expected to confirm the reception of every SDP packet. Otherwise the host assumes packet loss and retries.

C. Optimizations

While a SpiNNaker system can be comprised of a huge number of chips, every individual chip is rather limited in terms of memory storage and processing power. To compensate for these constraints, we introduced some minor modifications to the original HTM model which are briefly outlined in the following subsections.

1) Simplifications: Some of the details in [2] have been changed or omitted to account for the special properties of the SpiNNaker architecture. First, the inhibition step was simplified. In [2], columns have an inhibition radius. This means that a column will be activated by the spatial pooler if it is contained in the set of the columns with the highest overlap in its inhibition radius. To implement this, one would need to compute this set for every column in the region in every

inhibition step. Therefore the inhibition radius is omitted in our implementation, which is equivalent to having a global inhibition radius that contains all columns. This simplification also allows for a more efficient boosting mechanism. According to the original specification, the average activity of a column needs to be compared with the average activities of all other columns inside its inhibition radius. Since the radius is now global, it is sufficient to compute the maximum average activity of all columns only once per cycle and then compare this global maximum to a column's average activity in order to decide if boosting is required for that column.

To compute the accurate average activity of a column as proposed in [2], it is required to store the activity of a column over several time steps to determine the exact moving average. Our implementation for SpiNNaker uses an approximation to remove the need for keeping that much data. The new average activity is computed from the value of the last time step and the current activation state as follows:

$$a_{i+1} = a_i - (a_i - b_{i+1}) / w \tag{1}$$

a_i is the average activity in the i-th cycle, b_i is one if the column is active in cycle i and otherwise zero. The variable w is equivalent to the moving average window size. This approximation comes, of course, at the cost of precision.

2) Garbage Collector: The temporal memory learns novel patterns by creating new segments and connections. Since the HTM is an online unsupervised machine learning system, it should adapt to new and changing patterns and eventually unlearn old patterns [2]. As the unlearning of patterns works by negative reinforcement which leads to connections with permanences of zero, old and unused connections accumulate over time. This is essentially a memory leak that slows down the system since the temporal memory contains functions that iterate over all segments and connections. We addressed this issue by adding a garbage collector (GC) to our implementation. The GC works by checking the last cycle in which a segment was active and by removing the segment and all of its connections if the time stamp is old enough. On a more fine-grained level, the GC also inspects single connections and checks their timestamps. These connections can also be removed as soon as the assigned permanence value reaches zero. To make garbage collection possible, the temporal memory updates timestamps every time a segment or connection becomes active. The aggressiveness of the GC can be regulated through a parameter. In general, the GC improves performance and memory efficiency without impacting accuracy negatively. However, the properties of the input patterns must be considered carefully when setting this parameter. The longer and the more complex the pattern is, the longer are the intervals between consecutive activations of certain segments (connections). In such cases, a very aggressive GC is detrimental to the accuracy of the HTM as it will remove segments (connections) representing parts of a pattern that is currently being learned by the HTM.

D. Classification

Every cycle of an HTM ends with a set of cells in the predictive state, which is the output of the region [2]. This is a problem because a region can predict several inputs at once and the output set does not directly translate to one or more expected input patterns. Reconstructing possible inputs that match the prediction of the region is not a trivial task. In contrast, it is quite easy to determine if or how much a given input matched the region's prediction by simply checking all active columns and counting how many of them burst (i.e. did not expect the input) versus how many of a column's cells predicted their activation (i.e. had their prediction confirmed). A simple way to detect anomalies based on this idea is to calculate the ratio of bursting columns to all active columns after every cycle. If this ratio exceeds a threshold one can declare the input received in this cycle an anomaly. This solution proved to be sufficient for reliably detecting anomalies in patterns of low complexity and length.

IV. Evaluation

In order to evaluate the performance of our HTM implementation, several tests that measure the accuracy of the pattern recognition and anomaly detection were conducted based on the following protocol: Every test case consisted of five valid trials. Trials were considered invalid if the program did not terminate successfully or if the average amount of active columns did not reach a predefined threshold ranging from 0.5% to 1%, depending on the test case. This threshold was used because a region with very few active columns produces very unreliable results. The threshold was set before a test case was executed and consistently applied for every trial in the test case. For every test case, measurements were conducted until five valid trials were completed. In total, eight test were conducted, each measuring performance under a different combination of parameters and input properties.

A. Recorded Data

Four data points were collected for each test case: Pattern recognition rate (true patterns), anomaly detection rate (true anomalies), pattern failure rate (false anomalies) and anomaly failure rate (false patterns). The calculations were done in the following way:

$$r_i = \left(\sum_{j=1}^{5} a_{ij}\right) / \left(\sum_{j=1}^{5} b_{ij}\right) \tag{2}$$

r_i is the pattern recognition rate in the i-th test case, a_{ij} is the number of recognized patterns and b_{ij} is the number of given patterns in the j-th trial of the i-th test case. All other data points were calculated in the same manner. This method of averaging over all trials of a test case was chosen since the amount of given random input values (i.e. anomalies) could differ quite strongly among the trials. The reason for this is that in every time step a random number generator decided with a fixed probability if a random input value should be provided to the region instead of the next part of the pattern.

The pattern consisted of a string of integers, fixed in length, repeated in a circular way until the trial terminated. The integers ranged from 0 to 99. The probability of an anomaly occurring at an individual cycle was 5%. Each trial had a duration of 500 cycles, which means the region received 500 input values. Of those cycles, the first 100 were considered the warm-up period in which no data points were collected. Nevertheless, anomalies were also injected during the warm-up period. The classification of an input value as either pattern or anomaly was done by means of calculating the ratio of bursting

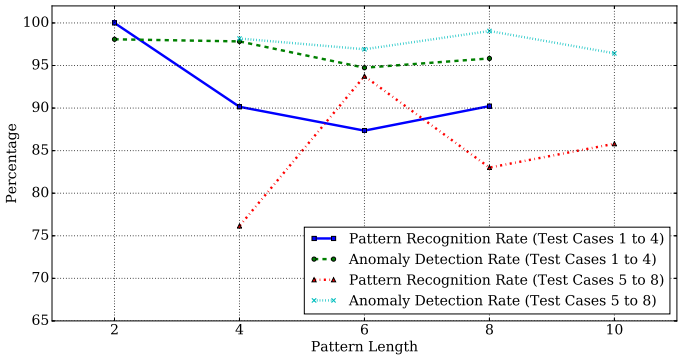

Fig. 3. Pattern recognition rate and anomaly detection rate versus pattern length. Test cases 1 – 4 used a region with 300 columns and 3 cells per column, test cases 5 – 8 a region with 500 columns and 5 cells per column.

to active columns. If the ratio exceeded a set threshold for an input value the input was classified as anomaly or as part of the pattern, respectively.

B. Results

The first three test cases with 300 columns and 3 cells per column show a steady decline of the pattern recognition rate (PRR) as the pattern length increases. The exception to this is the fourth test case, which shows an increase in performance compared to the third test case. This is presumably an artifact of the randomness of the HTM system and/or the low number of trials per test case. Test cases five to eight with 500 columns and 5 cells per column show a more sporadic behavior with no recognizable trend in PRR as the pattern length increases. Comparing test cases two to four and five to seven reveals that increasing the amount of columns and cells per column does not automatically improve PRR. In fact, the average PRR is worse in the test cases with more columns and cells per column but the same pattern length. All results are shown in Fig. 3.

In general, experimenting with the parameters of the HTM has shown that the HTM's behavior strongly depends on how the parameters are set. The parameters are also interdependent as the optimal values for a set of parameters could depend on the values of other parameters. These properties render finding the optimal parameter settings for a particular task a non-trivial problem. A manual trial and error approach is generally prone to getting stuck in a local optimum where changing some values and then gradually adapting the other values could lead to a better overall performance of the system.

V. Conclusion and Outlook

In this paper, we presented an implementation of the HTM model for the SpiNNaker neuromorphic architecture. While staying as close to the HTM specification as possible, constraints imposed by the SpiNNaker platform required a few minor modifications of the original algorithms. Nevertheless, an initial evaluation clearly proved that our implemented model is able to learn from input data to both predict future inputs and to detect anomalies.

However, at the current stage of development, the productive use is still quite limited. Setting the model parameters has to be done manually, which is not a viable solution for larger HTM regions and more complex input data sets. One possible solution could be the evolution of parameter sets via a genetic algorithm. As recently reported in [12], SpiNNaker can be used to speed up the learning process by running multiple individuals in parallel. Another constraint of the current implementation is the limitation of a region to a single chip. HTM models of practically relevant sizes will definitely need to span across several chips, which will require more advanced mechanisms for synchronization and data exchange within the system.

Adding the missing features described above will make the HTM model directly available to robotic systems that already have an interface to SpiNNaker. Moreover, we are considering to replace a workstation which is currently running the HTM model in a closed-loop setup with a biomimetic robot by a SpiNNaker system with our neuromorphic implementation of HTM.

Acknowledgment

This project has received funding from the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant Agreement No 720270 (Human Brain Project SGA1).

References

[1] Numenta, "HTM Cortical Learning Algorithms," 2011. [Online]. Available: https://numenta.org/resources/HTM_CorticalLearningAlgorithms.pdf

[2] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin, "Biological and Machine Intelligence (BAMI)," 2016. [Online]. Available: http://numenta.com/biological-and-machine-intelligence/

[3] F. Walter, F. Röhrbein, and A. Knoll, "Neuromorphic implementations of neurobiological learning algorithms for spiking neural networks," *Neurobiologically Inspired Robotics: Enhanced Autonomy through Neuromorphic Cognition*, vol. 72, pp. 152–167, 2015.

[4] A. K. Seth, O. Sporns, and J. L. Krichmar, "Neurorobotic models in neuroscience and neuroinformatics," *Neuroinformatics*, vol. 3, no. 3, pp. 167–170, 2005.

[5] K. Amunts, C. Ebell, J. Muller, M. Telefont, A. Knoll, and T. Lippert, "The Human Brain Project: Creating a European Research Infrastructure to Decode the Human Brain," *Neuron*, vol. 92, no. 3, pp. 574–581, 2016.

[6] A. Knoll and M.-O. Gewaltig, "Neurorobotics: A strategic pillar of the Human Brain Project," in *Brain-inspired intelligent robotics: The intersection of robotics and neuroscience*, J. Sanders and J. Oberst, Eds. Washington, DC: Science/AAAS, 2016, pp. 25–34.

[7] S. B. Furber, F. Galluppi, S. Temple, and L. A. Plana, "The SpiNNaker Project," *Proceedings of the IEEE*, vol. 102, no. 5, pp. 652–665, 2014.

[8] C. Richter, S. Jentzsch, R. Hostettler, J. A. Garrido, E. Ros, A. Knoll, F. Röhrbein, P. van der Smagt, and J. Conradt, "Musculoskeletal Robots: Scalability in Neural Control," *IEEE Robotics & Automation Magazine*, p. 1, 2016.

[9] J. Conradt, F. Galluppi, and T. C. Stewart, "Trainable sensorimotor mapping in a neuromorphic robot," *Emerging Spatial Competences: From Machine Perception to Sensorimotor Intelligence*, vol. 71, pp. 60–68, 2015.

[10] S. Billaudelle and S. Ahmad, "Porting HTM Models to the Heidelberg Neuromorphic Computing Platform." [Online]. Available: http://arxiv.org/pdf/1505.02142v2

[11] S. Temple, "SpiNNaker Datagram Protocol (SDP) Specification," 2011. [Online]. Available: https://spinnakermanchester.github.io/docs/spinn-app-4.pdf

[12] A. Vandesompele, F. Walter, and F. Röhrbein, "Neuro-Evolution of Spiking Neural Networks on SpiNNaker Neuromorphic Hardware," in *2016 IEEE Symposium Series on Computational Intelligence (IEEE SSCI 2016)*, 2016.