CORE
Metadata, citation and similar papers at core.ac.uk
Provided by PDXScholar

Portland State University
PDXScholar

Dissertations and Theses Dissertations and Theses

1-1-2011

Hierarchical Temporal Memory Cortical Learning Algorithm for Pattern Recognition on Multi-core Architectures

Ryan William Price
Portland State University

Let us know how access to this document benefits you.
Follow this and additional works at: http://pdxscholar.library.pdx.edu/open_access_etds

Recommended Citation
Price, Ryan William, "Hierarchical Temporal Memory Cortical Learning Algorithm for Pattern Recognition on Multi-core Architectures" (2011). *Dissertations and Theses.* Paper 202.

10.15760/etd.202

This Thesis is brought to you for free and open access. It has been accepted for inclusion in Dissertations and Theses by an authorized administrator of PDXScholar. For more information, please contact pdxscholar@pdx.edu.

Hierarchical Temporal Memory Cortical Learning Algorithm for Pattern Recognition on Multi-core Architectures

by

Ryan William Price

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science
in
Systems Science

Thesis Committee:
Dan Hammertsrom, Chair
George G. Lendaris
Christof Teuscher

Portland State University
©2011

Abstract

Strongly inspired by an understanding of mammalian cortical structure and function, the Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) is a promising new approach to problems of recognition and inference in space and time. Only a subset of the theoretical framework of this algorithm has been studied, but it is already clear that there is a need for more information about the performance of HTM CLA with real data and the associated computational costs. For the work presented here, a complete implementation of Numenta's current algorithm was done in C++. In validating the implementation, first and higher order sequence learning was briefly examined, as was algorithm behavior with noisy data doing simple pattern recognition. A pattern recognition task was created using sequences of handwritten digits and performance analysis of the sequential implementation was performed. The analysis indicates that the resulting rapid increase in computing load may impact algorithm scalability, which may, in turn, be an obstacle to widespread adoption of the algorithm. Two critical hotspots in the sequential code were identified and a parallelized version was developed using OpenMP multi-threading. Scalability analysis of the parallel implementation was performed on a state of the art multi-core computing platform. Modest speedup was readily achieved with straightforward parallelization. Parallelization on multi-core systems is an attractive choice for moderate sized applications, but significantly larger ones are likely to remain infeasible without more specialized hardware acceleration accompanied by optimizations to the algorithm.

Contents

Abstract **x**

List of Tables **xi**

List of Figures **xii**

1 Introduction **1**

1.1 Background 1

1.2 Systems Science Perspective 2

1.3 Key Aspects of Hierarchical Temporal Memory (HTM) 4

1.3.1 Encode Inputs Differently Depending Upon the Context . . 4

1.3.2 Sparse Distributed Representation 5

1.3.3 Hierarchy 6

1.4 Description of the Algorithm 6

1.4.1 Spatial Pooling Algorithm 9

1.4.2 Temporal Pooling Algorithm 10

1.4.2.1 Phase 1 11

1.4.2.2 Phase 2 12

1.4.2.3 Phase 3 13

2 Literature Review and Motivation **15**

2.1 Literature Review 15

2.2 Motivation 17

2.3 List of Contributions 19

3 Methodology **21**

3.1 General Overview 21

3.2 Need for Verification 22

3.3 Choice of Parallelization Method 23

4 Verification Testing and Investigation of Algorithm Properties **26**

4.1 Test Setup 27

4.1.1 Input 27

4.1.2 Network Parameters 28

4.1.3 Training 28

4.1.4 Testing 29

4.2 Learning First Order Sequences 29

4.3 Learning Higher Order Sequences 30

4.4 Algorithm Behavior with Noisy Data 33

5 Pattern Recognition Task for Performance Analysis **36**

6 Analysis of Sequential Implementation **40**

6.1 CPU Time of Sequential Implementation 40

6.2 Hotspots 42

6.2.1 segmentActive 43

6.2.2 getBestMatchingSegment 44

7 Parallelization of the Sequential Implementation **45**

7.1 Parallel Coverage 47

7.2 Load Imbalances 47

7.3 Result of Parallelization Effort 48

8 Discussion **50**

8.1 Theoretical Maximum to Parallel Scalability 51

8.2 Increasing Parallel Coverage to Improve Scalability 53

9 Conclusions and Future Work **56**

References **59**

A Software Guide **62**

A.1 Introduction 62

A.2 Requirements 62

A.3 Getting Started 63

A.3.1 Compiling 63

A.3.2 Setting Network Parameters 63

A.3.3 Datasets 64

A.3.4 Network Output 64

B Software Reference 65

B.1 data Namespace Reference 65

B.1.1 Variable Documentation 66

B.1.1.1 testfiles 66

B.1.1.2 trainfiles 66

B.2 Cell Class Reference 68

B.2.1 Member Function Documentation 69

B.2.1.1 setCell 69

B.3 Column Class Reference 70

B.3.1 Member Function Documentation 73

B.3.1.1 computeSPScore 73

B.3.1.2 infComputeScore 73

B.3.1.3 infIsActive 74

B.3.1.4 isActive 74

B.3.1.5 setCellNeighbors 74

B.3.1.6 setNeighbors 75

B.3.1.7 setSP 75

B.3.1.8 setTP 75

B.3.1.9 SPColumnLearning 76

B.3.1.10 updateDutyCycle 76

B.4 DendriteSegment Class Reference 78

B.5 Example Struct Reference 79

B.5.1 Detailed Description 79

B.5.2 Member Function Documentation 80

B.5.2.1 read_next 80

B.6 Level Class Reference 81

B.6.1 Constructor & Destructor Documentation 82

B.6.1.1 Level 82

B.6.2 Member Function Documentation 82

B.6.2.1 clearStates 82

B.6.2.2 generateData 83

B.6.2.3 generateData 83

B.6.2.4 generateOutput 83

B.6.2.5 printSS 83

B.6.2.6 printStates 84

B.6.2.7 SPinference 84

B.6.2.8 SPinhibition 84

B.6.2.9 SPlearning 84

B.6.2.10 SPoverlap 85

B.6.2.11 TPinference 85

B.6.2.12 TPlearn1st 85

B.6.2.13 TPlearning 85

B.6.2.14 updateStates 86

B.6.2.15 Verify2 86

B.7 Network Class Reference 87

B.7.1 Member Function Documentation 89

B.7.1.1 inference 89

B.8 segUpdate Struct Reference 90

B.8.1 Member Data Documentation 91

B.8.1.1 newSynapsesToAdd 91

B.8.1.2 segToUpdate 91

B.8.1.3 sequenceSegment 91

B.9 SpatialPooler Class Reference 92

B.9.1 Constructor & Destructor Documentation 92

B.9.1.1 SpatialPooler 92

B.9.2 Member Function Documentation 93
B.9.2.1 computeMatch (inf only) 93
B.9.2.2 computeMatch 93
B.9.2.3 increaseBoost 94
B.9.2.4 increasePermanences 94
B.9.2.5 UpdateActiveColPerm 95
B.10 TemporalPooler Class Reference 96
B.10.1 Constructor & Destructor Documentation 97
B.10.1.1 TemporalPooler 97
B.10.2 Member Function Documentation 98
B.10.2.1 adaptSegments 98
B.10.2.2 clearStates 99
B.10.2.3 ffOutput 99
B.10.2.4 getActiveSegment 99
B.10.2.5 getBestMatchingCell 100
B.10.2.6 getBestMatchingSegment 101
B.10.2.7 getCells 101
B.10.2.8 getSegmentActiveSynapses 102
B.10.2.9 infPhase1 103
B.10.2.10 infPhase2 103

B.10.2.11 phase1 104

B.10.2.12 phase1start 105

B.10.2.13 phase2 105

B.10.2.14 phase3 105

B.10.2.15 printSS 106

B.10.2.16 printStates 106

B.10.2.17 segmentActive 106

B.10.2.18 setLateralNeighbors 107

B.10.2.19 updateStates 107

B.11 example.cpp File Reference 108

B.12 htm.cpp File Reference 109

B.12.1 Detailed Description 109

B.13 params.h File Reference 109

B.13.1 Detailed Description 112

B.14 rng.h File Reference 112

B.14.1 Detailed Description 113

B.15 topology.h File Reference 113

B.15.1 Detailed Description 113

B.15.2 Typedef Documentation 114

B.15.2.1 actOvrPair 114

B.15.2.2 XYindex 114

B.16 tp.h File Reference 114

B.16.1 Typedef Documentation 115

B.16.1.1 activeSynapsePair 115

List of Tables

4.1 Network Parameters for Verification Tests 28

5.1 Digit Sequences for Pattern Recognition 38

5.2 Datasets for the Digit Sequence Recognition Task 39

7.1 Datasets Used for Measuring Parallel Scalability 45

List of Figures

1.1 Representation of a Column with its Proximal Dendrite 8

1.2 Representation of a Cell with its Distral Dendrites 9

1.3 Overview of Spatial Pooling 10

1.4 Overview of Temporal Pooling 11

1.5 Phase 1 of Temporal Pooling Algorithm 12

1.6 Phase 2 of Temporal Pooling Algorithm 14

1.7 Phase 3 of Temporal Pooling Algorithm 14

5.1 Binary Representation of a Digit from MNIST Database 37

6.1 Execution Time of Sequential Implementation 41

6.2 Normalized Execution Time of Sequential Implementation 42

6.3 Breakdown of Execution Time by Function 43

7.1 Parallel Scalability After Initial Parallelization 46

7.2 Speedup After Further Parallelization and Dynamic Scheduling . . . 49

8.1 Parallel Efficiency of Best Implementation 50

8.2 Theoretical and Actual Speedup Achieved 52

8.3 Effect of Increased Parallelization on Theoretical Speedup 55

Chapter 1

Introduction

1.1 Background

The Hierarchical Temporal Memory Cortical Learning Algorithm (HTM CLA) presents a unique and novel way of approaching problems in Machine Learning, Artificial Intelligence and Data Mining, amongst others. The development of the HTM CLA marks one of the most complete attempts to utilize knowledge of cortical structure and operation in a functional machine learning technology that is applicable to many problem domains. An HTM network can be considered a new form of neural network with a significantly more sophisticated model of the neuron. HTM is but one member in a family of biologically inspired, hierarchically organized network structures. Other members of this family include HMAX [1], Convolutional Neural Networks [2] and Deep Belief Networks [3], but the strong inspiration from mammalian cortex and potential application across a variety of problem domains places the HTM CLA at the forefront.

In between the bottom-up view of neuroscience and the top-down view of AI and statistical machine learning lies a variety of interesting behaviors such as perception, inference, prediction and complex movement. Neural Networks (NNs) lie somewhere in between these two approaches. They have been successfully employed in a wide variety of tasks and are capable of recognizing different kinds of patterns; the latter is a necessary (if not sufficient) aspect of intelligence. But

most neural network models have been extraordinarily simple in comparison to the massive complexity of the human brain which contains about one hundred billion neurons, each one connected to thousands of others [4]. NN models typically possess a very rudimentary neuron-like element and their lack of scalability to large implementations is a significant obstacle.

It is increasingly obvious that we need to understand biological intelligence in order to build machines that exhibit intelligence and there is no real alternative but to study the neocortex. Neuroscience has achieved a good understanding of the behavior of neurons and the functioning of many parts of the brain but lacks a theory of intelligence as a whole, leaving a wide gap in understanding. The HTM CLA may offer a new approach to bridging the large gap between our understanding of neural mechanisms and manifesting intelligent behavior in machines.

1.2 Systems Science Perspective

In contrast to, but not at odds with traditional sciences, systems science includes a strong focus on relations amongst "things" rather than just the things themselves. Systems science can be described as the "science of relations"; "systems problems" are problems of understanding relations; "systems knowledge" is essentially knowledge about relations. As systems scientists we approach problems by first abstracting away the "thingness" of a system and then seeking to understand the relations of the system. It is often useful to describe not only relations, but systems themselves, in terms of different levels. An investigator may define a system

with a set of relations which can be broken into sub-systems (or super-systems) with a different set of elements and relations. It is from this general, yet powerful perspective that systems science approaches scientific inquiry.

In systems literature Cartesian products are typically the basis for defining relations with a relation being a subset of some Cartesian product of given sets [5]. We also note that any mapping is a relation, though not all relations are a mapping. Nonetheless, it is sometimes useful to think about relations in terms of mappings. The brain receives sensory input patterns through time and forms relations between these input patterns and the 'real world' causes of them. At the highest level (call this the "A level") we observe that the brain performs some mapping of sensory input to output (be it behavioral, perceptual, inferential, etc.). The HTM CLA characterize a process for mapping sensory stimuli to specific cellular activation patterns that result in inference and prediction. Thus we expect that an HTM CLA simulation would produce outputs that indicate that handwritten digits, for instance, are being correctly classified after the system has been presented with adequate training examples.

At a conceptual level, NNs instantiate mappings from input to output and thus serve as valuable tools for approaching this problem and for the systems scientist in general. Some classes of NNs have an associated universal approximation theorem which suggests that (these types of) NNs are well suited to performing input-output mappings of a general nature. Therefore it is reasonable to believe that NNs are a practical vehicle for characterizing a process that maps inputs to outputs, and if formulated appropriately, do so in a manner similar to how the brain maps sensory inputs through time to outputs.

In order to see how the HTM CLA might produce the desired outputs at the A level, we need to shift perspective from the A level to a subsystem level (call it the "B level")[1]. The HTM CLA emulate cortical structure and functioning using relatively simple, local rules. At the B level cortical columns and their corresponding cells interact with feed-forward stimuli as well as locally with each other. The work of this project is done at this B level. I have implemented the algorithm that carrys out these simple, local rules and verified that the implementation is functioning properly. We suspect that the B level interactions defined by these algorithms will result in the desired mapping being observed at the A level but do not attempt to validate that the behavior of the model actually corresponds to the behavior of the brain. It would be interesting to investigate the theoretical nature of how such an A level mapping emerges from the B level emulation of the cortex but that is beyond the scope of this project.

1.3 Key Aspects of Hierarchical Temporal Memory (HTM)

In this section I will briefly discuss some of the key aspects of HTM.

1.3.1 Encode Inputs Differently Depending Upon the Context

The HTM CLA provide a method for representing the same input differently depending upon the context of previous inputs. The brain must have a way of forming different internal representations for the same sensory input when it is preceded by different sequences of inputs. This is a universal feature of perception and action

[1] See "On Systemsness and the Problem Solver: Tutorial Comments" by Lendaris for a full discussion on the use of different perceptual levels to aid problem solving [6].

[7]. The role a previous input context plays in representing and recognizing input sequences is discussed more in chapter 4.

1.3.2 Efficient Encoding Using a Sparse Distributed Representation

Information in the brain is represented as a sparse distributed representation (SDR). Much like actual neurons in the brain, HTM cells are highly interconnected but local inhibition ensures that only a small percentage are active at any one time. Though the number of possible input patters is much greater than the number of possible representations, forming a SDR of the input does not generate a practical loss of information. In fact it has several advantageous properties.

When used in conjunction with an appropriate storage algorithm, SDR possesses the property of mapping similar inputs to similar representations. Because the number of possible representations is often much greater than the actual number of representations used, only a subset of the input patterns need be matched to guarantee a correct match. The similarity of two patterns can be effectively identified by comparing the overlap of bits (in the case of a bit string).

Perhaps the most advantageous property of SDR is efficiency. SDR is memory efficient because it provides an encoding that allows you to store a number of unique inputs that is far larger than the number of representing units. It is also computationally efficient. To really take advantage of the increasingly large amounts of data available we need to utilize the efficiencies provided by SDR.

1.3.3 Hierarchy

A constant theme in almost all cortical circuitry is hierarchy. As in the cortex, information processing in an HTM network is hierarchical. At the lowest level of an HTM network the input patterns are constantly changing, much like the incoming sensory stimuli we humans receive. Traveling up the hierarchy, spatial and temporal resolution dilate. Cell activation patterns are more stable because information is transferred up the hierarchy in predictable sequences. The brain constantly compares incoming sensory patterns and stores a model of the world that is largely independent from how it is perceived under changing conditions. To accomplish this the cortex forms invariant representations at all levels in a hierarchy.

Hierarchical structure can aid in the modeling of high dimensional input spaces with moderate amounts of memory and processing. Hierarchy also significantly improves efficiency in that it reduces training time and the amount of memory required. This is, in part, because low-level patterns are recombined at the mid-levels of the hierarchy, and mid-level patterns are recombined at high-levels. To learn a new high level pattern you don't need to relearn all of its components [7]. It also leads to more efficient use of neuron connections, perhaps the biggest cost in implementing such algorithms in hardware.

1.4 Description of the Algorithm

In an HTM CLA network, SDR is used to learn a large number of spatial patterns and temporal sequences. Training data in the form of an input stream is presented

to the network and a model of the statistical structure of the training data is built. Unlike models for static pattern recognition, HTM accounts for spatial *and* temporal variability in the input data. It accomplishes this by learning sequences of commonly occurring input patterns in an unsupervised manner.

In explaining HTM some definitions are in order. The term "layer" is common to both neural network terminology and neuroscience. Here "layer" carries the neuroscience connotation and all the layers in a cortical sheet are modeled by a "level". "Column" and "cell" are closely related to the corresponding neuroscience terms. A column is an organizing element of the cortex and consists of a large number of cells. "Region", also carries the neuroscience connotation with HTM regions containing interconnected cells arranged in columns. Several regions can exist at the same level and be arranged in a hierarchy.

An HTM region is made up of columns, each of which contains interconnected cells (see figure 1.1). Cells have both feed-forward and lateral inputs via proximal and distal dendrites respectively. All cells in a column share a single proximal dendrite with an associated set of potential synapses which map a subset of the input space to a given column. Feed-forward input may come from sensory data or from another level lower in the hierarchy. Synapses are not fixed and have the ability to connect or disconnect through time based on a "permanence" value.

Cells may have many distal dendrite segments each of which also has an associated set of potential synapses (see figure 1.2). The set of potential synapses are mapped to a subset of other cells within a neighborhood[2], also called a "learning radius".

[2] A cell's "neighborhood" refers to the other cells within a certain radius around it but does not include the other cells in the same column to which it belongs.

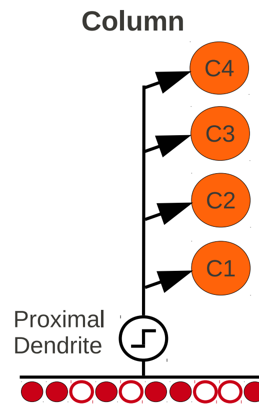

Figure 1.1: A column with four cells is depicted. The cells in a column share a common proximal dendrite which maps to the input space or the immediately lower level via a set of synapses which are depicted as a set of circles in red at the bottom. Solid circles represent a valid synapse connection to an input bit with a permenance value above the connection threshold. Non-solid circles represent a potential synapse connection to an input bit with a permenance value below the connection threshold. Feed-forward input may result in a column becoming activated after a local inhibition step if enough valid synapses are connected to active input bits.

A dendrite segment forms connections to cells that were active together at a point in time, thus remembering the activation state of other cells in the neighborhood. If the same cellular activation pattern is encountered again by one of its segments, i.e., the number of active synapses on any segment is above a threshold, the cell will enter a predictive state indicating that feed-forward input is expected to result in column activation soon.

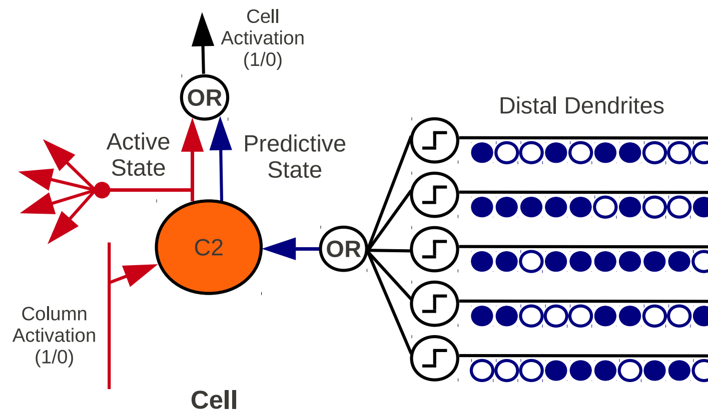

Figure 1.2: A cell is depicted with its distral dendrites, shown to the right. Each dendrite segment has several synapse connections to other cells within its learning radius. Solid (blue) circles represent a valid synapse connection to another cell with a permenance value above the connection threshold. Non-solid circles represent a potential synapse connection to another cell with a permenance value below the connection threshold. Column activation resulting from feed-forward input via the proximal dendrite is shown in the bottom-left. Cells in a column share a single binary-valued column activation signal. Individual cells have their own binary-valued "active" state that participates in the feed-forward output of a cell and is also propagated to other cells via lateral connections depicted in the upper-left. The cell may enter a "predictive" state if at least one of its dendrite segments is connected to enough active cells. A cell's binary-valued predictive state only participates in the feed-forward output of a cell and is not propagated laterally. The cell outputs the boolean OR of its active state and predictive state to the next level.

1.4.1 Spatial Pooling Algorithm

Starting with sensory input, a sum is computed by convolving input data in a column's receptive field with the set of associated synapses (i.e., its proximal dendrite). A column's sum is multiplied by a scalar "boost" value. Columns which habitually have a low sum after the convolution step are given a larger boost. Boosting is designed to promote relatively uniform activity among the columns. An

inhibition step follows in which columns with a strong activation inhibit columns with a weaker activation within the local neighborhood. The local inhibition results in a sparse set of active columns that serves as input for the temporal learning phase at that same level. In the active columns, Hebbian like learning is used to strengthen synapses that were aligned with active input and weaken synapses aligned with inactive inputs. Synapses whose permanence value exceeds or falls below a threshold value will become valid or invalid accordingly.

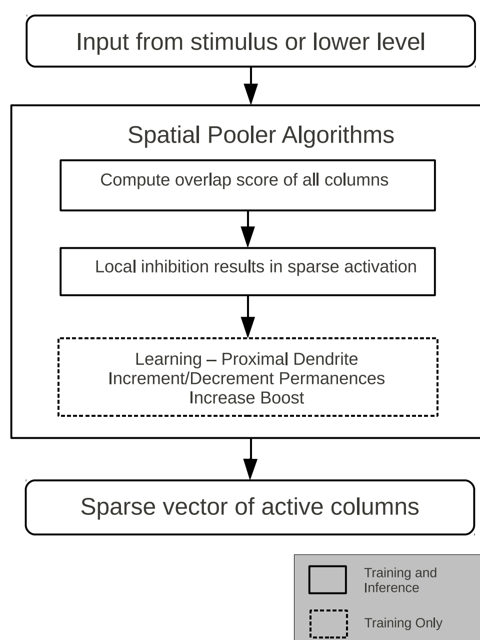

Figure 1.3: Overview of Spatial Pooling.

1.4.2 Temporal Pooling Algorithm

It is convenient to organize the temporal pooling algorithm into 3 phases. Portions of phases 1 and 2 are performed while a network is learning as well as during inference. Phase 3 is performed during learning only. A similar organization of the

algorithm is employed in [7], which may serve as a useful reference. The phases are described in the sections that follow.

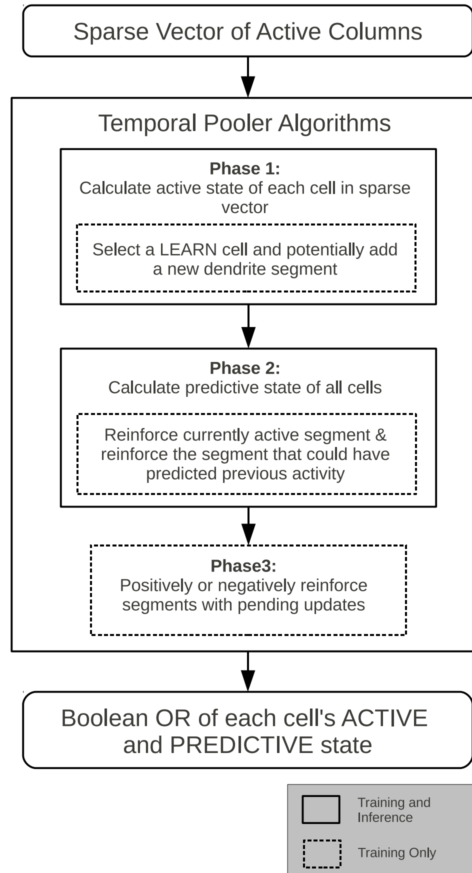

Figure 1.4: Overview of Temporal Pooling.

1.4.2.1 Phase 1

When a column becomes active due to feed-forward input, it first checks to see if any of its cells are in a predictive state from a previous time step, meaning that the current activation was anticipated. If a cell was predicting the current input, then that cell is switched from predictive to active. The resulting set of all active cells

represents the current input in the context of the previous input. If no cells were predictive then the input was not anticipated and *all* cells in the column are set to active. Furthermore, the cell that has the dendrite segment that best matches the input at the previous time step is selected for learning.

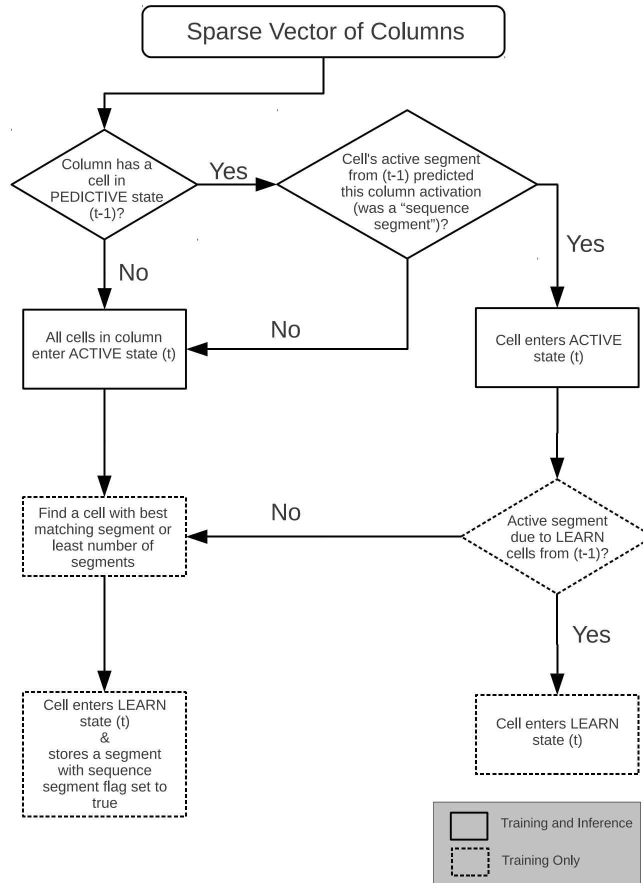

Figure 1.5: Phase 1 of Temporal Pooling Algorithm.

1.4.2.2 Phase 2

Alternatively, cells in any column may enter a predictive state. Every dendrite segment on every cell is checked to see if the number of active synapses connected

to currently active cells is above the threshold. If it is, the dendrite segment is activated and the cell enters a predictive state. Similar to the synapses of the proximal dendrite, whenever a dendrite segment becomes active, the permanence values of its associated synapses are modified according to the Hebbian rule. However, these changes are marked as 'temporary' until we will know if the cell correctly predicted the feed-forward input, at which point changes in permanence values will either be removed or allowed. In addition to the modifications to the synapses associated with the active segment, the cell's segment that best matches the state of the system at the previous time step is also selected for learning in order to predict sequences further back in time. Using the previous state of the system, the permanence values of its associated synapses are modified according to the Hebbian rule and are also marked as 'temporary'. Finally, a vector representing the active and predictive states of all cells in the level becomes the input to the next level in the hierarchy.

1.4.2.3 Phase 3

Cells which have undergone learning have pending changes to existing dendrite segments and may also have learned new segments. If the cell correctly predicts feed-forward input, then these pending changes are made permanent and the permanence values of the appropriate synapses are incremented. Otherwise, if the cell ever stops predicting, then these pending changes are cleared and the permanence values of the appropriate synapses are decremented.

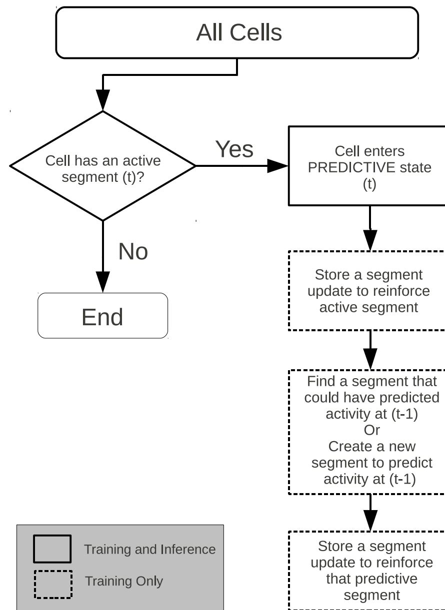

Figure 1.6: Phase 2 of Temporal Pooling Algorithm.

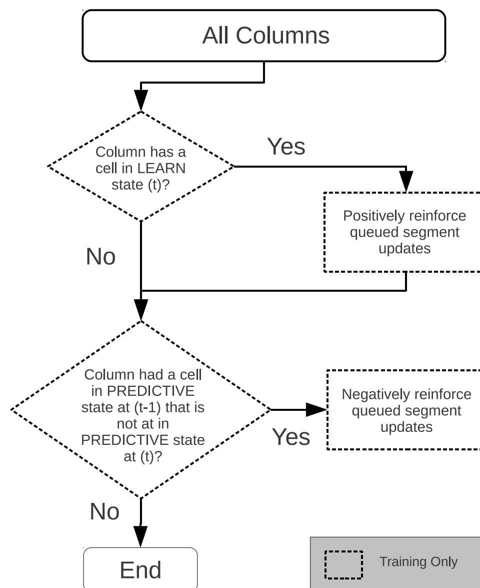

Figure 1.7: Phase 3 of Temporal Pooling Algorithm.

Chapter 2

Literature Review and Motivation

2.1 Literature Review

Hawkins' theory of the brain as a memory system and the basis for what he has called the "memory-prediction system" were first laid out in a book he co-authored called "On Intelligence" [8]. A mathematical framework was developed by George [9]. The theoretical concepts, mathematical framework and biological mapping were in continuous development for a number of years by Numenta, a California based company[10]. Additionally, others studied HTM applications [11, 12] and several commercially successful applications were developed.

The prior versions of the HTM algorithms differ significantly from the HTM CLA. Prior versions of the algorithm used Markov chains and Bayesian Belief Propagation. In these versions, novel input patterns were compared to the subset of stored input patterns and the likelihood over the set of stored input patterns was calculated. The likelihood over the set of stored patterns became the input to the temporal learning component of a node in which a Markov graph of temporal transitions was learned by building a first-order transition matrix. The Markov graph was then partitioned to form Markov chains. The likelihood over the spatial input pattern was used to compute the single most probable Markov chain given the current evidence. The most probable Markov chain was passed as input to the

next layer of nodes. With the development of the HTM CLA, Numenta discontinued further research of these earlier versions. There is no evidence whether others continue to pursue them.

Several other notable models have taken a cue from neuroscience and utilize hierarchical structure with common neural elements to represent sensory information and capture spatiotemporal dependencies. The Deep SpatioTemporal Inference Network (DeSTIN) is a type of deep learning architecture that combines unsupervised learning for dynamic pattern representation with Bayesian inference. Every node in the architecture has a common functionality and the belief states formed across the hierarchy inherently capture sequences of patterns, and spatiotemporal dependencies within the data. This approach shares some similarities with previous versions of HTM algorithms but uses a discriminative, rather than a generative model [13].

Chappelier and Grumbach explored a connectionist architecture that handles spatiotemporal patterns[14]. The RST (réseau spatio temporal) network takes into account spatial and temporal aspects at the architectural level of the network. The spatial aspect is addressed by a specific connection distribution function and the temporal aspect is addressed via a leaky-integrator neuron model with a refractory period and postsynaptic potentials. Numerous other temporal connectionist models exist and constitute a growing body of research in temporal processing with neural networks [15].

2.2 Motivation

I believe that an accurate and scalable model for predicting sequential data would be instrumental in overcoming a number of existing challenges. Important potential applications for this type of model include the identification of objects in images and video, the identification of a speaker in an audio recording, control signals for machines, resource management in complex systems, web analytics, power use optimization, and the prediction of power system failure. The HTM CLA are actually the result of continuous algorithm development over several different versions in the last 5 years by Numenta. The HTM CLA version of the algorithms is a significant departure from previous versions. In a technical report recently made available on their website, Numenta describes the theoretical framework for the algorithms and provides psuedocode [7].

The HTM CLA represent perhaps the most rigorous attempt to date to model the general structure and function of the neocortex in a machine learning algorithm. While the algorithms are not mathematically sophisticated, they are of considerable procedural complexity. Not surprisingly, a full implementation of HTM CLA is a significant commitment of time and effort, but such an effort is essential for further analysis of the algorithms and for determining potential improvements. A published study of HTM CLA performance on an actual data set has not yet been done. Accordingly, the computational costs of the HTM CLA are also unknown, as well as the performance on multi-core architectures. However, there is some concern that the computational costs could impede the wide-spread adoption of the HTM CLA. A need exists for more study of the performance of HTM CLA

with real data and the associated computational costs.

Power limitations constrain faster clocks and so performance improvements now have to come from parallelism. The semiconductor industry is moving into ever larger numbers of multiple cores, but unlike faster clock speeds, which were transparent to the program, programmers now need to ensure their applications are designed to be able to do many tasks in parallel which can be a difficult proposition. Simultaneously, massive amounts of data are becoming available for use in machine learning applications. To really take advantage of high performance computing and ever larger amounts of data, we need to exploit the available parallelism. A multi-core implementation will offer important insights into the ability to accelerate the HTM CLA using a common computer architecture. The use of multi-core architectures represents just one of several high performance computing platforms, but they are the most generic and are a good place to start.

This research is guided by the following questions:

1. What kind of execution time can be expected when using the HTM CLA for a pattern recognition task?
2. Will it scale well with larger amounts of data?
3. How does the HTM CLA scale on a multi-core system?

There are numerous implementation decisions associated with the HTM CLA and this is an opportunity to explore the implementation space and begin addressing the many implementation and operation questions that remain. By implementing the algorithm as it is described by Numenta, my goal has been to understand

what kind of performance can be expected from the HTM CLA and at what computational cost. Also, having an implementation of the "standard" version of the algorithms is essential to further research efforts. I am not attempting to verify that the algorithms accurately model the behavior of the mammalian cortex, or attempting to show that they can outperform other machine learning technologies in some pattern recognition task.

The algorithm appears to lend itself well to parallelization and a corollary goal has been to see if significant speed up is possible using a multi-core architecture. The aim has been to develop a full implementation of the HTM CLA and verify proper functionality using a test process described by Numenta. The project reported here developed and implemented a parallelized version for use on multi-core architectures, and was compared to the benchmark established by the sequential version. Performance of the multi-core implementation can be used in a comparative study exploring more specialized hardware like GPUs and FPGAs which is likely to follow as future work.

2.3 List of Contributions

This work resulted in the following contributions:

1. A complete and verified C++ implementation of the current version of the HTM CLA available for use in future research projects including machine learning applications and continued algorithm development.

2. A performance analysis of a parallel implementation of the HTM CLA on a multi-core system that will be used in a comparative study with other

hardware including FPGA and GPUs. Additionally, an estimate of scalability based on further parallelization is provided.

3. The identification of two subroutines, segmentActive and getBestMatchingSegment, that together account for more than 90% of the total execution time. These two subroutines are key to an HTM CLA acceleration effort.

4. The identification of two high-level HTM CLA routines, Phase 2 and Phase 1 of the temporal pooling algorithm, that are responsible for all segmentActive and getBestMatchingSegment calls.

5. The measurements of sequential execution time using five sizes of data sets provide a reasonable estimate of HTM CLA sequential implementation performance in a representative pattern recognition task.

6. Detailed software documentation provided makes this complete implementation more accessible to users and may aid other developers in their own implementation.

7. A discussion of observations made during the implementation verification process that provides insight into the nature of HTM CLA first order and higher order sequence learning.

8. A first look at HTM CLA behavior with noisy data using a simple experiment.

Chapter 3

Methodology

3.1 General Overview

The research was conducted in two phases.

The first phase comprised:

1. Implement a single process version of the full HTM CLA in C++.
2. Verify proper functioning of the implementation using verification tests described by Numenta.
3. Design a pattern recognition task suitable for analysis of the sequential and parallel implementations.
4. Benchmark the sequential implementation on the pattern recognition task using Intel's VTune parallel workbench and program analysis package.

   The second phase comprised:

5. Identify key hotspots to focus parallelization efforts.
6. Implement a parallel version of the code.
7. Analyze the parallel version running on multiple cores using VTune.
8. Perform a parallel scalability analysis of the multi-core trials.

The PC used for analyzing the implementation has a Intel Xeon X5650 6-core CPU with 12GB RAM.

The primary focus of this work is on implementation and hardware mapping aspects, not on the recognition results of the algorithm. Attempting to build the best classifier using this algorithm would have added additional complexity to an already complex project, and would distract from the stated motivations. It would also be beyond the scope of a single MS thesis. However, applications of the algorithms need to be explored and this project resulted in a functioning implementation that will allow us to pursue future research in applications and in hardware implementation. Furthermore, use of VTune focused on identifying hotspots in the sequential code in order to guide the parallelization effort. No attempt was made to modify the algorithms in order to improve performance or explore other trade-offs. The HTM CLA is a complex, newly developing algorithm and in many ways it is still a "moving target". There are many potential modifications to the algorithms that may be explored but are beyond the scope of this work.

3.2 Need for Verification

C++ was selected as the programming language for the implementation because of its fast execution speed and because it is one of the few languages supported by the two APIs considered for implementing parallelization. Whatever the chosen language is for an implementation of the HTM CLA, verification of the implementation should be performed to ensure that the subtleties of the algorithm have been

well understood and that the implementation functions as intended. The verification tests described to us by Numenta ensured proper functioning and clarified the more subtle aspects of the algorithms. The assurance of proper functioning and better understanding of the algorithms gained by the verification process became even more valuable in light of the second phase of this project. Parallelization adds another level of complexity and it is important to have sequential code that has been well tested and debugged before parallelizing such code.

3.3 Choice of Parallelization Method

There is an ongoing discussion in the high performance computing (HPC) community on the topic of how to best approach parallel programming on multi-core systems. Two distinct approaches to parallel programming were considered for this work, multi-threading and message passing, each of which offers its own advantages and disadvantages. It is generally accepted that multi-threading provides a quick, efficient approach for shared memory parallel programming and that message passing is intended for distributed memory systems, but it can also be used on multi-core systems and frequently is. A hybrid approach using both message passing and multi-threading may achieve greater results than either approach used in isolation, but it presents a considerable challenge to the programmer who is not well-experienced in HPC programming.

OpenMP is a multi-threading API for multi-platform shared-memory parallel programming. More specifically, OpenMP is a set of compiler directives and library routines that extend C++ (as well as C and Fortran). Shared-memory parallel

programs created through OpenMP are executed by multiple independent threads on one or more processors that share some or all of the available memory. The API provides a means for starting up threads, assigning work to them and coordinating synchronization. Implementing parallelism using OpenMP is often straightforward once the programmer has identified where the parallelism is in the program. Though not always the case, significant performance gains may often be achieved with OpenMP by using basic compiler directives and expecting the compiler to generate the parallel code. *Using OpenMP* [16] by Chapman, Jost and Van Der Pas is a valuable resource for information on OpenMP and shared memory parallel programming.

Unlike the shared-memory model of parallel programming, message passing assumes each process will have its own private address space. Message passing libraries, such as MPICH2, are based on the Message Passing Interface (MPI), a specification for message passing libraries. MPICH2 and other such libraries provide a means for initiating and managing each process, as well as operations for sending and receiving messages between processes. Although the original message passing model implies that processes will exchange messages whenever one of them needs data from another one, "MPI-2", the newer MPI specification, extends the original model to include "single-sided communication" which allows a process to directly access memory in another process without needing to call any corresponding send or receive operation in the other process. *Using MPI-2: Advanced Features of the Message Passing Interface* [17] by Gropp, Lusk, and Thakur is a good reference for information on message passing and single-sided communication.

OpenMP possess several very attractive qualities which were key in the decision

to use OpenMP for the parallelization effort instead of an MPI library. OpenMP is a smaller API and the set of features needed to do simple parallelization can be learned quickly. After identifying where the parallelism lies in a program, OpenMP can be applied incrementally to parallelize the program by inserting directives into a sequential program and letting the compiler determine the details of the parallel code. Once the additional code has been compiled and tested, another portion of code can be parallelized from the sequential code. This process does not require a single major reorganization of the sequential code as is typical of MPI in which it's "all or nothing." Incidentally, the application can still compile as sequential code even on a compiler that has no knowledge of the OpenMP standard. The remote memory operations specified by MPI-2 are powerful but should be distinguished from the shared-memory model employed by OpenMP because the address space is not shared so programs cannot be conveniently written using the familiar variable reference and assignment statements as they can in the shared memory model [16]. In summation, OpenMP was found to be easy to learn, offered a smooth, incremental approach to parallelization without a lot of reorganization, and conveniently handled variables of complex user-defined data types.

Chapter 4

Verification Testing and Investigation of Algorithm Properties

Due to the complex nature of the algorithm and its implementation, particularly the parts associated with temporal pooling, specialized testing was necessary to verify proper functionality of the implementation. A series of verification tests were suggested by Numenta upon request. These tests are not a simple comparison of accuracy results on a data set. They required additional time and effort in terms of code writing and debugging, but provided a much higher level of confidence that the most complex pieces of the algorithm are implemented correctly. In addition to verification of the implementation, these tests also provide some insight into the behavior of the algorithms.

The verification tests focus on the temporal pooling operation. Unlike the spatial pooling operation, whose functionality is relatively easy to observe and verify during a typical debugging process, the temporal pooling algorithm can quickly become too difficult for someone to verify during typical step by step debugging process and cannot easily be confirmed from the final output of the network. The verification process can be divided into two categories of tests: the first relates to learning a first order sequence using a single cell per column instantiation. It also explores how many and what size sequences can be learned in a simple one cell per column network. The second category involves learning higher order sequences using multiple cells per column and could be used to better understand why a multi-celled configuration is essential for learning higher order sequences.

An additional test was conducted after the verification tests were completed to study how the algorithm behaves with noisy data. This test is described later in section 4.4. Next, we will discuss the verification tests, beginning with a general description of the test setup used for these tests before describing the specifics of each test.

4.1 Test Setup

4.1.1 Input

M input sequences, each consisting of N random patterns, are used. Each 100 bit pattern contains between 21-25 active bits. The active bits of each pattern are selected randomly subject to the constraint that a sequence does not contain any consecutive patterns with a common active bit.

As an example consider the following valid sequence of 3 patterns, each of length 10:

0110100110
1001011000
0100100110

The following example is not valid because it has two consecutive patterns (the 1st and 2nd) that both contain active bits in the 2nd and 5th (from the left) position.

0110100110
1101100001
0010011110

4.1.2 Network Parameters

A 10 by 10 array of columns is used, with one column per input bit. Cells are capable of forming a synapse connection to any other cell in the network[1]. When forming a dendrite segment, 11 cells out of the 21-25 active columns are randomly chosen for forming synapse connections[2] . While determining cellular activation states, at least 9 of the 11 synapses in a dendrite segment must be active for the segment to be considered active[3]. The minimum threshold for learning is set to 11 synapses, ensuring that new dendrite segments are learned each time and no additional synapses are added to existing segments. These parameters and others are summarized in table 4.1.

Table 4.1: Network Parameters for Verification Tests

Parameter	Value
New Synapse Count	11
Activation Threshold	9
Minimum Threshold	11
Initial Permanence	80
Connected Permanence	70
Permanence Decrement	0
Permanence Increment	40
Maximum Permanence	100

4.1.3 Training

Training is done with P passes of the M sequences, presenting each of the N patterns one at a time. This makes the total number of iterations during training equal to

[1]Cells may not connect to other cells in the same column when a multiple cell per column network is used

[2]This is determined by the "New Synapse Count" parameter

[3]This is determined by the "Activation Threshold" parameter

P*N*M. Cellular activation patterns are cleared between sequences by reseting the network. Only strict sequence learning is tested during the verification process. As a result, the part of the Phase 2 temporal pooling algorithm that learns dendrite segments in order to predict more than one time step into the future is disabled for all verification tests.

4.1.4 Testing

Learning is disabled and the same set of sequences is presented to the network for inference. Again the network is reset after each sequence. The network should accurately predict the next pattern at each time step up to and including the N-1st time step for each sequence. A prediction is considered perfect if every column in the prediction is correct and no extra columns are in a predictive state. If 2 or more columns are incorrect in a given prediction, the test failed.

4.2 Learning First Order Sequences

Networks with one cell per column are used to learn first order sequences. Prediction of first order sequences does not require any temporal information. When doing first order predictions, inference is based only on the static recognition of the current input pattern. In other words, only the current input pattern is used to predict the next input pattern. With a first order network (a network with one cell per column), a given input pattern will always result in the same prediction being made by the network, regardless of the other inputs that preceded it. The reason why a first order network can't learn higher order sequences is discussed

further at the end of the next section.

Test F1 Test that a first order sequence can be learned with M=1, N=100, P=1.

Test F2 Same as Test F1, except P=2. The same sequence is presented twice and we check that synapse permanences are incremented and that no additional synapses or segments are learned. The test fails if additional synapses or segments are learned during the second pass.

Test F3 See how many sequences can be learned with N=300 and P=1. The network was able to learn one 300-pattern sequence passing the test. When two sequences were learned the network incorrectly predicted 4 patterns.

Test F4 See how many patterns can be learned by varying N and M. What is the largest possible value of N*M? Start with N=100, M=3, P=1. The largest value of N*M achieved was 375 with N=125 M=3. Runs with N=100 M=4 and N=150 M=3 both incorrectly predicted 2 patterns.

4.3 Learning Higher Order Sequences

In contrast to first order networks, which make predictions based only on the current input, higher order networks (networks with multiple cells per column) are capable of utilizing variable length context to learn time-based sequences. In higher order sequences, the same spatial pattern may appear in several different contexts and so information beyond the current input is necessary for prediction.

This set of tests verifies that high order sequences can be properly learned in a multiple cells per column configuration. The parameters are the same as the first

order tests but multiple cells per column are used for some of the tests. No special training or test procedures aside from those described are required for the higher order sequence tests, but generating higher order input sequences does require an additional constraint. In addition to the conditions described previously, the sequences are constructed to contain shared subsequences. Consider two sequences of 10 input patterns, where each input pattern is represented by a letter:

A B C D E F G H I J
K L M D E F N O P Q

The subsequence DEF is made up of three consecutive patterns that appear in both sequences. The position and length of shared subsequences are parameters in the tests. Two sequences of 100 patterns containing a shared subsequence of 8 patterns (the 50th through 57th patterns) were used.

Test H1 Two sequences with a short shared subsequence are learned using a network with one cell per column. The same parameters from B1 are used (M=2, N=100, P=1). This test should fail because only one cell per column was used and multiple cells per column are required to learn these types of sequences.

Test H2 Run test H1 again but with four cells per column. This test should pass.

Test H3 Run test H2 again with P=2. Check that synapse permanences are incremented and that no additional synapses or segments are learned. The test fails if additional synapses or segments are learned during the second pass.

In order to investigate the process in which a first order network fails during higher order sequence prediction, detailed output from tests H1 and H2 was examined closely. These tests consisted of sequences of 100 patterns with a shared subsequence of 8 patterns but for ease of discussion I make analogy to the two sequences of 10 input patterns (represented by letters) with a shared subsequence of three input patterns (DEF) previously given as an example. I will begin by describing the observations made during the training of the first order network before moving on to the higher order network.

Training of the first order network proceeds in the following manner. New segments are learned at each time step (starting with the second) while the first sequence is presented. As the second sequence is presented new dendrite segments are learned until the start of the shared subsequence, pattern D. A representation of input pattern D was learned from the first sequence and when this pattern is encountered again it triggers a correct prediction of E and no new dendrite segments are learned at the next time step, instead the appropriate synapses are reinforced. This process repeats when pattern E precedes pattern F again. The network then predicts that pattern G will follow F but the novel input pattern N appears instead and new dendrite segments are learned. Learning new segments proceeds until the end of the sequence. Because the network now has learned to represent pattern F as preceding both G and N, whenever either of the two sequences containing F is presented the network will predict that both G and N follow F instead of one or the other.

Next we will examine training of the higher order network with four cells per column. New dendrite segments are learned at each time step through the first

sequence. However, unlike the first order network, new dendrite segments will continue to be learned throughout the second sequence, even when the input patterns of the shared subsequence are encountered. When the start of the shared subsequence, pattern D, appears it results in the same set of active columns after spatial pooling (due to the feed forward stimulus being exactly the same). However, the cellular activation of these actives columns will not be same because each column is capable of having any combination of it's four cells in an active state. Thus while columnar activation for input pattern D is the same as it was when pattern D was encountered in the first sequence, the cellular activation is not the same due to the different context from prior inputs. In contrast, a first order network only has one cell, thus the similar columnar activation will always result in the same cellular activation. This observed difference in the training of first versus higher order networks provides some insight as to why the two types of networks behave differently when noisy input sequences are encountered.

4.4 Algorithm Behavior with Noisy Data

How do the HTM CLA behave when presented with a noisy sequence? If the network has correctly learned to predict a sequence of patterns and then is presented with a slightly erroneous copy of the sequence, will it recover quickly after any unexpected noisy patterns are encountered and correctly predict the rest of the sequence? This would be a desirable property since real world data is likely to be noisy. A simple test using a ten digit sequence of handwritten characters[4] was

[4]Each example digit used in the test was taken from the MNIST database, a database of handwritten characters created by Yann LeCun. After being converted to binary, each example is presented to the network as a 784 bit vector. Detailed information about the MNIST database

created to see how the HTM CLA might perform with a noisy sequence. After learning to correctly predict the sequence 0 5 9 1 3 7 4 2 6 8, the network was presented with the sequence 0 5 9 1 **2** 7 4 2 6 8 in which the 5th pattern, '3', was replaced with a copy of the 8th pattern, '2'. The behavior of both first order and higher order networks was studied.

As for the inference of the first order network, correct predictions are made for '5' and '9' after which '3' is incorrectly predicted to follow the '1'. Next, '6' is incorrectly predicted to follow the unexpected '2'. Following this, '7' is presented and results in the correct prediction of '4'. At this point the network has recovered and continues predicting the rest of the sequence correctly. It is interesting to note the incorrect prediction of '6' following the '2'. As explained in the previous two sections, this prediction is due to the first order network only being able to learn a first order memory of the input pattern '6' (namely, that '2' precedes it). This is not the case for a higher order network which is described next.

As in the first order network, '5' and '9' are correctly predicted then '3' is predicted after the '1'. The next time step brings a '2' and the network does not make a prediction, that is to say no cells enter a predictive state. The network has learned to predict a '6' following a '2' when '2' appears as the 8th pattern in the original sequence, however the patterns preceding this '2' are different so '6' is not predicted. In other words, the context of the prior input is not the same as the original sequence so this '2' is not mistaken for the '2' that precedes '6' in the original sequence. Following the '2' a '7' is presented and the network recovers as it did in the case of the first order network, accurately predicting '4' and then the

is provided in chapter 5.

rest of the sequence.

This test suggests that when presented with a sequence containing an erroneous pattern, the network will continue to predict the rest of the sequence correctly. The behavior of a first order network does differ from that of a higher order network when subjected to a simple noisy sequence. This difference makes sense in light of our understanding of how single and multiple cell per column networks learn to represent patterns.

Chapter 5

Pattern Recognition Task for Performance Analysis

While the verification tests described in sections 4.2 and 4.3 were essential in ensuring that the most difficult portions of the algorithm were functioning properly, they were inadequate for estimating implementation performance. Each verification test executed quickly and did not fully employ the spatial pooling algorithm. Because the verification tests could not be used to gain an adequate estimate of the implementation performance, a pattern recognition task was devised in order to provide a more representative baseline of implementation performance. The pattern recognition task makes use of the MNIST dataset made available by Yann LeCun which is actually a subset of a larger set available from the National Institute of Standards and Technology (NIST). It has a training set of 60,000 example digits, and a test set of 10,000 example digits. According to LeCun's website, the original binary images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images are 8 bit greyscale. The images were centered in a 28x28 image by computing the center of mass of the pixels, and translating the image so as to position this point at the center of the 28x28 field [18]. The images were converted back to binary for our use here and are presented to the network as a 784 bit vector of binary pixel values.

A typical pattern recognition task using this dataset could be characterized as presenting training examples of each digit one at a time during the learning phase, then testing the generalization of the classifier by presenting each test example and

Figure 5.1: Binary Representation of a Digit from MNIST Database

seeing how many digits are classified correctly. However, using the dataset in this manner does not incorporate any temporal information in the task and therefore would not serve as a representative baseline of the HTM CLA. Another alternative would be to create short 'movies' of each digit by presenting a digit as a series of translations, rotations and scales throughout the input field. However, such a sophisticated scheme is not necessary to create a temporal data sequence. A much simpler approach is to create a sequence of digits and train the network to recognize sequences of digits as opposed to individual digits. This task incorporates both spatial and temporal elements and is easy to conduct. To this end, 10 unique sequences, each consisting of the 10 digits, were created.

Table 5.1: Digit Sequences for Pattern Recognition

0591374268
4086927351
3792608145
7342605918
7089542613
6472389501
1682537904
3271804695
7895460312
2936814570

Note that some sequences contained shared subsequences of digits. For example,

70**89**542613
64723**895**01

This ensured that a higher order network would be necessary to adequately learn to represent the data as discussed in sections 4.2 and 4.3. After the 10 sequences of digits had been determined, MatLab code was written to generate five sizes of data sets (see table 5.2) for measuring the execution time of the sequential implementation. Examples of individual digits were selected without replacement from the MNIST database and were assembled to form examples of the digit sequences. Thus, each example sequence contains unique example digits. The MNIST dataset is large enough that 5,000 training sequences and 810 test sequences can be assembled for this pattern recognition task.

Finally, a note on parameter tuning. To achieve the best generalization results in a pattern recognition task, network parameters are often tuned, either by hand or through optimization, until a satisfactory set of parameters are found. My experience with these algorithms in their current form suggests that parameter selection

Table 5.2: Datasets for the Digit Sequence Recognition Task

Data Set	Train Sequences	Test Sequences	Total Iterations
1	200	50	2500
2	500	100	6000
3	750	130	8300
4	1000	150	11500
5	1250	180	14300

may strongly influence execution time, particularly parameters associated with the temporal pooler. This work does not study how well the network generalizes in the given pattern recognition task and so no attempt was made to adjust parameter settings in order to achieve the best generalization results. Nonetheless, I believe this pattern recognition task serves as a representative task from which to obtain performance measurements associated with scalability and that the network parameters used are a reasonable starting point if one were to actually apply the network as a classifier to the task.

Chapter 6

Analysis of Sequential Implementation

6.1 CPU Time of Sequential Implementation

Intel's VTune [19], a powerful threading and performance profiler for understanding an application's serial and parallel behavior to improve performance and scalability, was used to profile the sequential implementation run on a single core and establish a baseline for comparison. The CPU time was measured across the five data sets shown in table 5.2 with network parameters kept constant. Figure 6.1 shows a considerable increase in execution time as the size of the dataset increases with the largest data set taking well over 3 hours to complete.

While 3 hours per run may not be prohibitive for some applications, there is a large set of network parameters that will likely need to be tuned. In some applications, the data set may be significantly larger (for comparison, the MNIST database contains 70,000 examples in its entirety) and may have a greater number of dimensions. If the baseline measurements observed here are indicative of general HTM CLA performance, then in cases of larger, high-dimensional data, the run time of the HTM CLA is likely to be prohibitive.

If the execution time increased linearly with the number of iterations the network performs, then we would expect the CPU time per iteration to be constant. However, figure 6.2 shows that network iterations actually take longer to complete as the size of the data set increases and that the increase in CPU time is not due

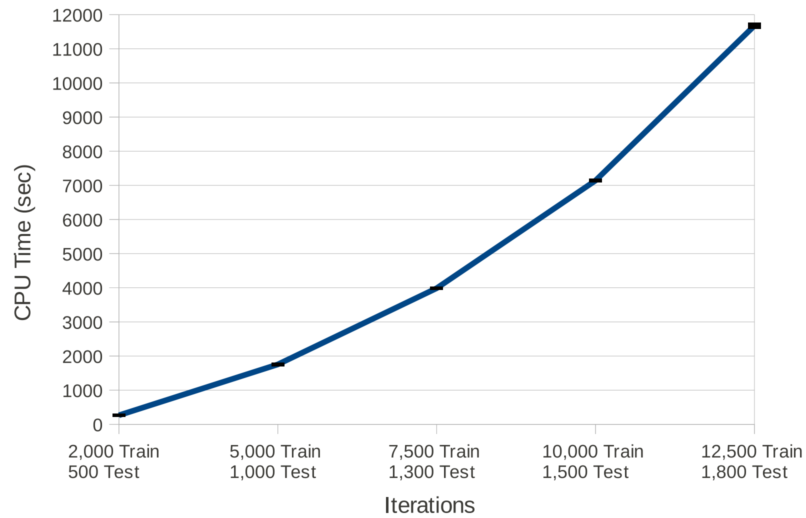

Figure 6.1: The CPU Time of the sequential implementation measured in seconds using the five sizes of data sets is shown. The considerable increase in execution time seen may be indicative of poor scalability with larger amounts of data and larger networks. Measurements are averaged over several runs and y-axis error bars display the 95% confidence interval (the intervals are so small they appear solid).

to simply running more iterations of a constant execution time. In other words, the CPU time per iteration does not remain constant as the size of the data set increases, it gets worse as the size of the data set increases.

Assuming that we have created a representative task for the HTM CLA with appropriate network parameters for the task, it is likely that the baseline measurements observed indicate that the algorithms would strongly benefit from speedup and the parallelization effort on a multi-core system is justified. Though an algorithm analysis of the HTM CLA is not done in this work, we expect that results of such an analysis would be consistent with our empirical results.

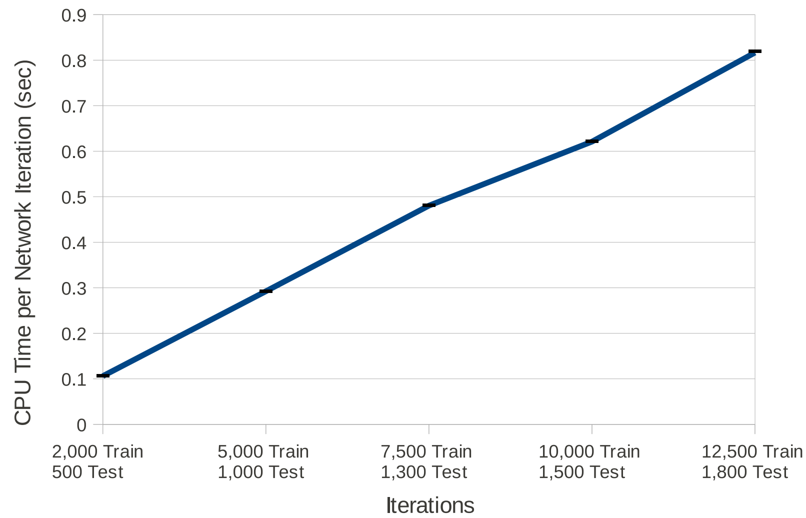

Figure 6.2: The average CPU time per iteration is shown for the five sizes of data sets. Instead of staying constant, the average CPU time per iteration increases as the size of the data increases. Iterations are taking longer to complete when the network has a larger data set to contend with. Measurements are averaged over several runs and y-axis error bars display the 95% confidence interval (the intervals are so small they appear solid).

6.2 Hotspots

Hotspots, code regions in the application that consume a lot of CPU time, were identified using the profiling data for three of the data sets collected by VTune. This analysis was an important step in guiding the parallelization effort, because it identified several functions in the algorithm that consumed considerable CPU time. We suspected that temporal pooling functions would make up the majority of CPU time, which they did. We did not expect to find that two temporal pooling functions would completely dominate (see figure 6.3). This was a significant discovery because it strongly influenced the approach to parallelization and led to significant performance gains with a simple, effective parallelization approach.

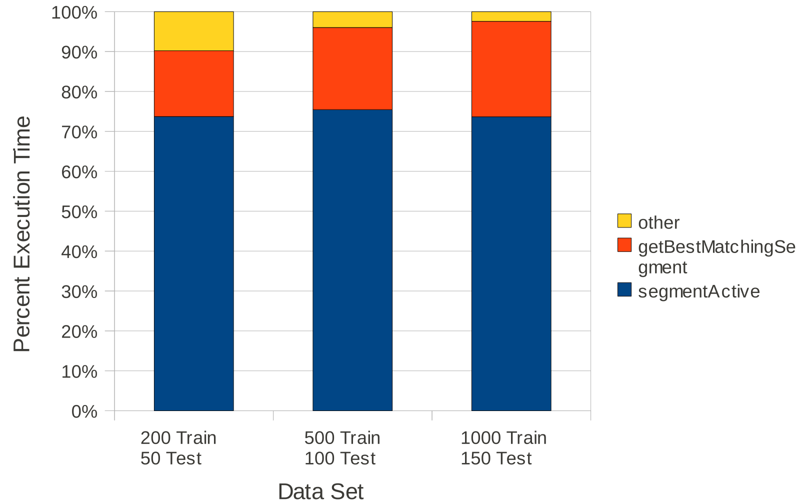

Figure 6.3: Using profiling data from the 2000 Train−500 Test, 5000 Train−1000 Test, and 10000 Train−1500 Test data sets, two hotspots were identified. CPU time is dominated by two sub-routines, segmentActive and getBestMatchingSegment, which take between 90%−98% of the total execution time. These two sub-routines are key to the parallelization effort.

Two temporal pooling functions, 'segmentActive' and 'getBestMatchingSegment' accounted for approximately 90% to 98% of the total execution time depending on the size of the data set. We observe that these two functions account for an increasing amount of the total execution time as the size of the data set increases, further underscoring the importance of targeting them for parallelization.

6.2.1 segmentActive

For a given dendrite segment, cell state and time, the segmentActive routine determines if the number of connected synapses is above the activation threshold. Profiling data from the 500 train and 100 test sequence data set run was organized by function and call stack to determine where the most time is spent on

segmentActive. Phase 2, described in figure 1.6, was the largest calling routine of segmentActive with 92% of segmentActive's CPU time attributed to Phase 2 calls. The segmentActive function executes quickly but accounts for such a large percentage of total CPU time because it is called many times. During Phase 2, segmentActive is called many times by every cell in every column. The number of segmentActive calls becomes large as more and more dendrite segments are learned with each training iteration. In a two dimensional network with a 28 by 28 region of columns, each having four cells, if each cell where to learn a 1000 dendrite segments, Phase 2 may result in upwards of 3 million segmentActive calls. Parallelizing segmentActive would probably not be worth the associated thread overhead. Instead, it makes more sense to parallelize the calling routine, Phase 2, effectively bringing the parallelization to the column level rather than the segment level.

6.2.2 getBestMatchingSegment

For a given cell and time, this routine finds the dendrite segment with largest number of active synapses. If the cell does not have any dendrite segments with enough active synapses above a minimum threshold, no segment is returned. Phase 2 was also found to be the largest caller of getBestMatchingSegment with just over 50% of getBestMatchingSegment's CPU time attributed to Phase 2 calls. Phase 2 was the obvious choice for starting an incremental parallelization approach. Phase 2 calls were responsible for the majority of segmentActive and getBestMatchingSegment's CPU time, which were in turn responsible for a large majority of the total execution time.

Chapter 7

Parallelization of the Sequential Implementation

Two hotspots, segmentActive and getBestMatchingSegment predominately executed in Phase 2, were identified using the profiling data. Consequently, Phase 2 of the temporal pooling algorithm was selected as the starting point for incremental parallelization. An initial parallelization of most of Phase 2 was carried out using loop-parallelization. The program was executed with up to 6 cores with three sizes of data sets shown in table 7.1, and profiling data was collected using VTune. Parallel speedup S, which is defined as $S_p = \frac{T_1}{T_p}$ where p is the number of processors, T_1 is the execution time of the sequential code and T_p is the execution time of the parallelized code with p processors, was calculated for each of the runs. Despite a large remaining fraction of sequential code and load imbalances, reasonable speedup was readily achieved with this simple and effective parallelization step. The results of this initial parallelization are shown in figure 7.1.

Table 7.1: Datasets Used for Measuring Parallel Scalability

Data Set	Train Sequences	Test Sequences	Total Iterations
1	200	50	2500
2	500	100	6000
4	1000	150	11500

Somewhat surprisingly, the largest of the three data sets did not benefit the most from the initial parallelization. This is likely due to an increase in primary memory access. With the larger data set, more dendrite segments are stored by each cell,

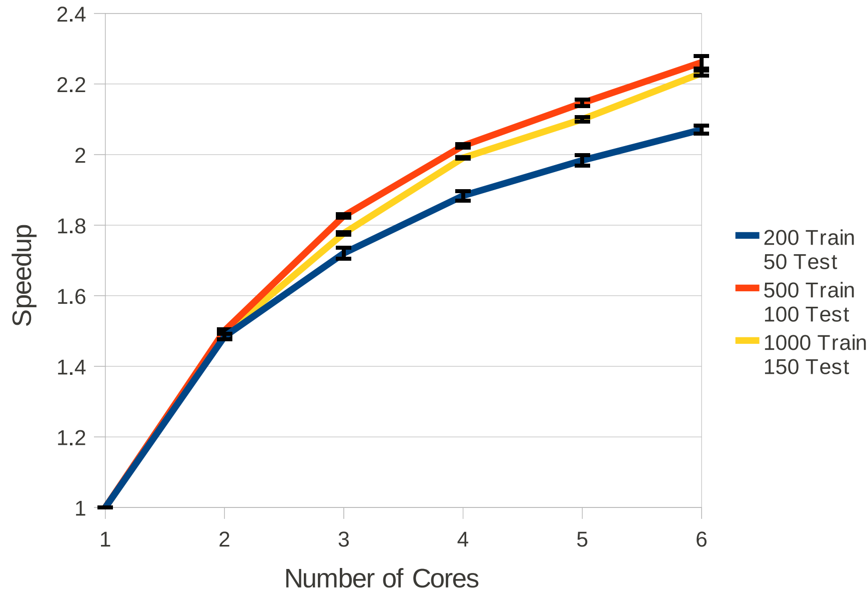

Figure 7.1: After some initial parallelization of the sequential code, execution time was measured with up to six cores using three of the data sets and speedup was calculated. Some speedup is readily achieved through loop-parallelization of Phase 2 of the temporal pooling algorithm which targets the identified hotspots, segmentActive and getBestMatchingSegment. Measurements are averaged over several runs and y-axis error bars display the 95% confidence interval (some intervals are so small they appear solid).

making each column larger and probably resulting in a decreased ability to effectively leverage the memory hierarchy. Profiling data shows that the percentage of execution time spent in parallel regions did increase with the larger data set but this was most likely offset by the penalty resulting from an increase in primary memory access. An analysis of memory access is needed to say conclusively.

7.1 Parallel Coverage[1]

Amdahl's Law, discussed in more detail in chapter 8, indicates that parallel scalability[2] is limited by the size of the sequential code remaining in a parallel program. An effort to increase the fraction of parallel code and reduce the fraction of sequential code was made through further parallelization. The remaining Phase 2 calls that had not yet been parallelized were brought into parallel regions, making parallelization of Phase 2 complete and further increasing speedup. Parallelization of Phase 1 of the temporal pooling algorithm (shown in figure 1.5), the routine which accounted for the remainder of segmentActive and getBestMatchingSegment calls, was parallelized at the cell level. However, preliminary results indicated that the benefit was outweighed by high overhead costs when Phase 1 was parallelized at the cell level. The parallelization of Phase 1 at the column level, which is how Phase 2 is parallelized, should be successful and is left for future work.

7.2 Load Imbalances

When threads perform different amounts of work in a work-shared region[3], threads with less work will finish faster and have to wait for the slower ones to finish and reach the synchronization barrier. Idle threads could be used to do other work. This uneven distribution of workload amongst threads is known as load imbalance and it can result in a significant performance hit. Though the HTM CLA is

[1] Parallel coverage is defined as the fraction of execution time spent inside parallel regions.

[2] Parallel scalability refers to a program's ability to decrease execution time with an increasing number of processors.

[3] Here a "region" refers to all the code encountered during a specific instance of the execution of a given section of code, including any called routines. Thus a work-sharing region is a given region of code in which the work is distributed among the executing threads.

designed to learn and store activation patterns in an evenly distributed number of segments across the region of columns, in practice there may be a large discrepancy in the number of dendrite segments stored amongst different cells in the region. If thread scheduling[4] is not done, some threads may be assigned cells with a small number of dendrite segments and others may be assigned cells with a large number of dendrite segments, leading to load imbalance.

OpenMP parallelized "for loops" are scheduled with static scheduling by default, meaning that each thread is assigned an equal number of iterations to complete. If there are n iterations and T threads, each thread will get n/T iterations[5]. It is possible to specify other scheduling schemes using a scheduling clause[6]. With dynamic scheduling, each thread executes a number of iterations specified by a chunk-size parameter. A "chunk" refers to a specific number of contiguous iterations that are allocated to a thread at a time. After a thread has finished executing a chunk of iterations, it requests another chunk, and continues until all of the iterations are completed[7][16].

7.3 Result of Parallelization Effort

After completing parallelization of Phase 2 and using dynamic scheduling to address load imbalance, parallel speedup was again measured using up to six cores with with the 500 Train, 100 Test size data set. Figure 7.2 shows notable improvement in speedup was achieved compared to the initial results, indicating that the

[4]Thread scheduling refers to the way threads are assigned to run on the available processors.

[5]OpenMP also handles the case when n is not evenly divisible by T

[6]Clauses may be appended to OpenMP directives for additional control over data sharing, synchronization and scheduling.

[7]The last set of iterations may be less than chunk-size

reduction in sequential and load imbalance overheads was significant.

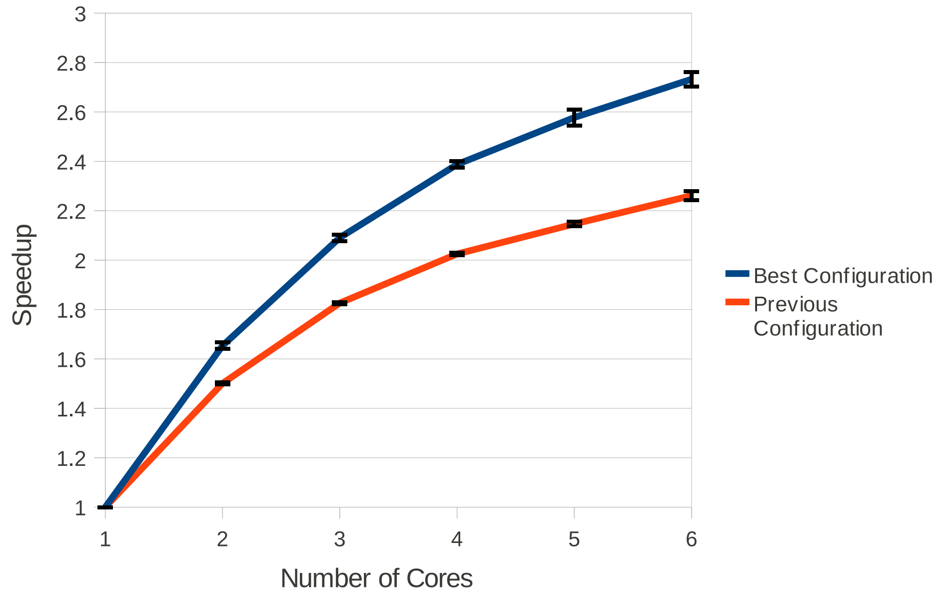

Figure 7.2: After additional parallelization and the use of dynamic scheduling, execution time was measured with up to six cores using the 500 Train, 100 Test dataset and speedup was calculated. Speedup for the previous parallel configuration is shown for comparison. Performance has improved notably but speedup remains modest. Measurements are averaged over several runs and y-axis error bars display the 95% confidence interval (some intervals are so small they appear solid).

Chapter 8

Discussion

Figures 7.1 and 7.2 show a continuing increase in speedup as the number of cores increases up to the 6-core case investigated here. This indicates that the limits to scalability have not yet been reached at six cores. However, the cores become increasingly less efficient as more are added, as seen in figure 8.1, and there are clearly diminishing returns. If the observed trend continues, we expect that the limits of scalability would be reached after adding but a few more cores. The maximum parallel speedup of this implementation would likely be around a factor of 3 with no significant additional increase seen with the further addition of cores.

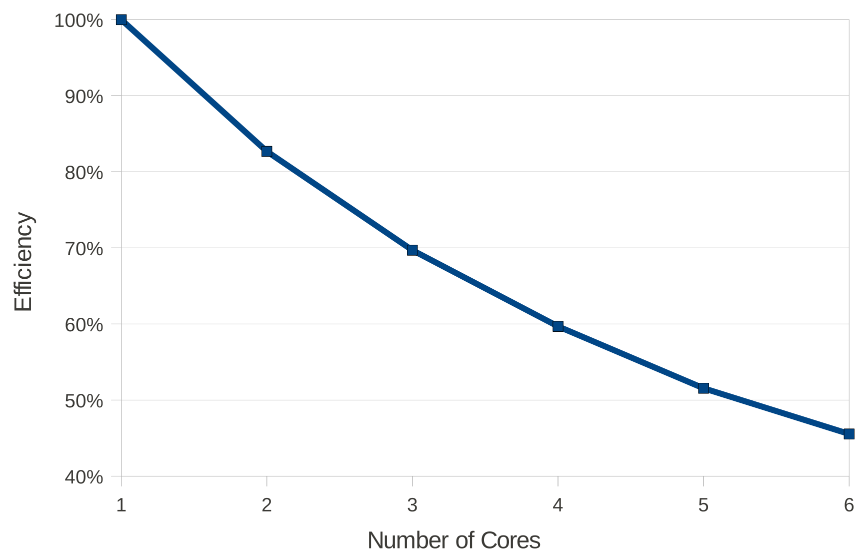

Figure 8.1: Parallel efficiency of the best configuration with the 500 Train, 100 Test size data set decreases as additional cores are added.

Though more aggressive parallelization of the implementation is possible, reasonable speedup was readily achieved with straightforward OpenMP directives and the execution time of the 500 Train, 100 Test data set was reduced from about 29 minutes to under 13 minutes. Based on the profiling data collected, we estimate that less than 25% of this time is spent performing inference on the test data. Therefore, performing inference using a test set and network of similar size should only take a few minutes after the network has been fully trained and that may be sufficient in many cases. If the speedup observed here is sufficient for a given HTM CLA application, then the multi-core approach is an attractive choice using common hardware. However, the scalability observed is considerably more limited than we had hoped for. In general, several factors can limit the scalability of a multi-threaded application. These factors include the fraction of sequential code, access to primary memory, parallelization overhead, load imbalance, and synchronization overhead [16]. However, synchronization overhead was not a factor in this implementation and load imbalance was addressed using dynamic scheduling, leaving the fraction of sequential code, access to primary memory and parallelization overhead as the likely causes of the limited scalability observed with this implementation.

8.1 Theoretical Maximum to Parallel Scalability

A theoretical maximum to parallel scalability is determined by Amdahl's Law[1] which is defined as $S = \frac{1}{(1-f_{par})+\frac{f_{par}}{P}}$ where f_{par} is the fraction of parallel code

[1] It's possible to do better than Amdahl's Law. Superlinear speedup may be achieved when a program has access to more cache and less data has to be fetched from main memory at run time.

and P is the number of processors. Given a certain fraction of sequential code, if a parallel program did not have any overhead, then the speedup defined by Amdahl's Law should be observed. The limit to scalability would be purely due to the remaining sequential code. The actual speedup observed in our parallel implementation is less than the theoretical speedup given the same fraction of sequential code, indicating that there is some overhead present that is causing a performance hit.

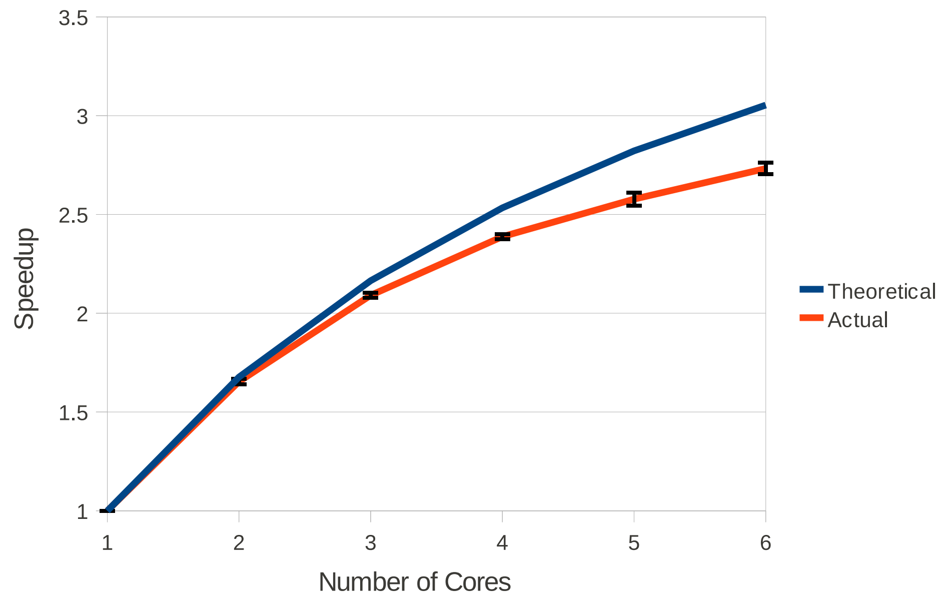

Figure 8.2: The actual speedup achieved with the best parallel configuration and 500 Train, 100 Test data set is compared to the theoretical limit determined by Amdahl's Law. Performance is slightly below the theoretical value, which is believed to be due to some remaining parallelization overhead and a performance penalty resulting from accessing primary memory. However, the theoretical speedup curve is considerably less than linear and the fraction of sequential code must be reduced to improve scalability. Measurements for the actual speedup curve are averaged over several runs and y-axis error bars display the 95% confidence interval (some intervals are so small they appear solid).

The theoretical speedup seen in figure 8.2 is significantly less than linear, showing that even without the presence of overhead, the scalability of this implementation is strongly limited by the size of the serial sections remaining[2]. More aggressive parallelization will be needed to keep serial code from limiting parallel scalability as the number of cores increases. With regard to the overhead observed, we suspect the obstacles to achieving the theoretical speedup are the overheads introduced by forking and joining threads and memory accesses. It appears that the increase in aggregate cache capacity provided by the additional cores was not enough to benefit from but an analysis of memory access by threads would be needed to say conclusively. That said, even if overhead was not causing a performance hit and near theoretical speedup could be achieved, it may still not be sufficient for many applications of the algorithm. Increasing the fraction of parallel code in the implementation appears to offer the greatest potential for improving scalability and achieving greater speedup on a multi-core system.

8.2 Increasing Parallel Coverage to Improve Scalability

This parallel implementation has targeted Phase 2 of the temporal pooling algorithm, which was found to be the largest consumer of CPU time by far. However, the algorithm is massively parallel and additional opportunities for parallelization

[2]A small part of this is due to the code associated with reading input data from a file and writing the network's output to a file. These functions were not considered for parallelization because they are not part of the core algorithms and their impact is likely to change depending upon the application. They accounted for about 1.4% of the total execution time for the 500 Train, 100 Test data set and therefore contribute to some of the sequential overhead that limits scalability.

exist. Phase 1 of the temporal pooling algorithm accounts for a much smaller percentage of execution time than Phase 2 does, so incremental parallelization should start with phase 2. Although, Phase 1 is responsible for a much smaller fraction of CPU time than Phase 2, it is responsible for a much greater percentage of CPU time than any of the other remaining routines. If both Phase 1 and Phase 2 where parallelized, scalability would likely improve greatly because the fraction of remaining sequential code would be relatively small. Using profiling data from the 500 Train, 100 Test data set runs, the theoretical speedup was calculated based on what the fraction of sequential code would be if both Phase 1 and Phase 2 were parallelized and if only Phase 1 were parallelized. It is compared with the theoretical speedup calculated when just Phase 2 is parallelized in figure 8.3.

There is no fundamental reason why this level of parallel coverage could not be realized in an implementation of the HTM CLA. Of course, we don't now what the actual speedup achieved by such an implementation would be and some overhead will surely be present. Nonetheless, we believe increasing parallel coverage would have a substantial impact on scalability. Figure 8.3 suggests that a considerable increase in speedup may be achieved with further parallelization of the remaining sequential code. However, it is not clear that even linear speedup would be sufficient when the data set and network are large, unless a large number of cores could be used. Other ways to accelerate the algorithm, or modifications to the algorithm, may need to be explored when significantly larger data sets and network sizes are required.

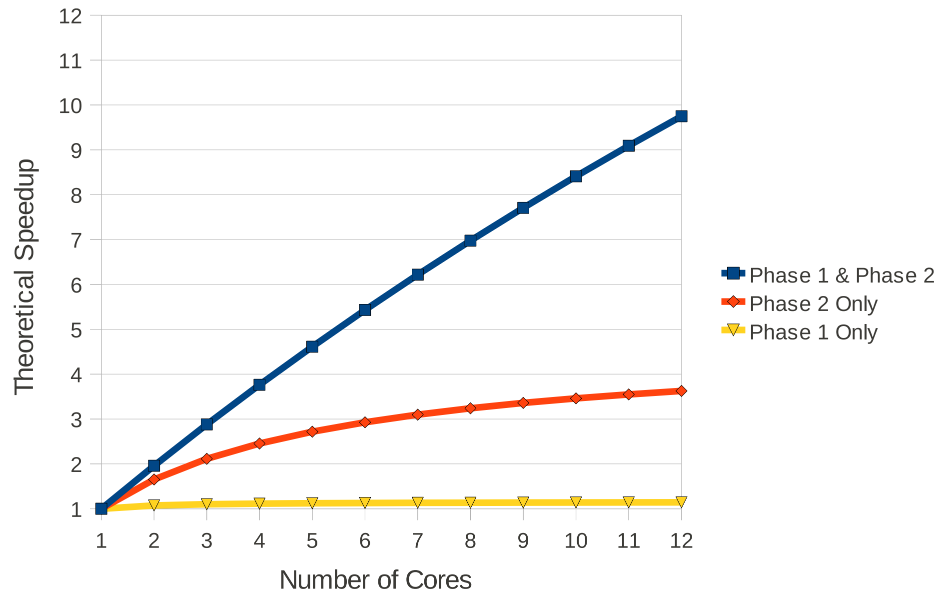

Figure 8.3: Theoretical speedup is calculated for three parallel configurations based on execution times from the runs with the 500 Train, 100 Test data set: complete parallelization of Phase 1 and Phase 2 of the temporal pooling algorithm, parallelization of Phase 2 only, and parallelization of Phase 1 only. Phase 1 is responsible for a much smaller percentage of total execution time than Phase 2 is, but scalability greatly improves when both Phase 1 and Phase 2 are parallelized.

Chapter 9

Conclusions and Future Work

Performance analysis of the sequential implementation in a pattern recognition task shows a rapid increase in execution time as the size of the data set increases and indicates that the performance problems may limit scalability which may be an obstacle to their adoption. The parallelized version developed for a multi-core system using multi-threading demonstrated that speedup is readily achieved with straightforward OpenMP directives that do not require major modifications to the sequential code. More aggressive parallelization than what was performed here is possible, but even without it we believe that parallelization on multi-core systems is a reasonable choice for moderate sized HTM CLA applications. However, the resulting speedup was modest (up to a factor of 3) and larger applications are likely to remain infeasible without further acceleration or modifications to the algorithm. Additional parallelism remains to be leveraged and analysis indicates that considerably better speedup may be achieved with the additional parallelization of Phase 1 of the temporal pooling algorithm but this is left for future work.

Any attempt to accelerate the HTM CLA should focus on the hotspots clearly identified as a result of the analysis in section 6.2. These two sub-routines, segmentActive and getBestMatchingSegment, are shown to be responsible for an increasingly large majority of the execution time, up to 98% of the total execution time. Furthermore, Phase 2 of the temporal pooling algorithm is a good place to start the parallelization effort since the majority of these two sub-routines' CPU

time was attributed to Phase 2 calls. Phase 1 of the temporal pooling algorithms accounts for the remainder of these two hotspots' CPU time and should also be targeted for parallelization.

Much remains to be discovered about the HTM CLA, which offers a novel approach to pattern recognition and inference in spatio-temporal problems. By employing what we believe to be a representative pattern recognition task and selecting reasonable network parameters, we have begun to understand what kind of execution time can be expected when using the HTM CLA for a pattern recognition task and how an implementation of the algorithm will scale with larger amounts of data. As seen in section 6.1, execution time for some of the larger data sets was on the order of several hours. Likewise, the parallel version has informed us as to how the HTM CLA scales on a multi-core system and what kind of speedup can be expected. The parallelization results described in section 7.3 indicate that speedup of around a factor of three can be expected when only Phase 2 is parallelized, but theoretical calculations in section 8.2 suggest that much greater performance may be achieved by parallelizing both Phase 2 and Phase 1. Though not a primary focus of the work, some aspects of the algorithms themselves were investigated. First order and higher order sequence learning were briefly examined during verification of the implementation in section 4.3 and algorithm behavior with noisy data was examined in a simple experiment in section 4.4.

Many opportunities for future work remain. In order to get the best possible parallelization results from a multi-core implementation, the remaining sequential fraction of code should be parallelized. Future work could address the remaining sequential code through more aggressive parallelization. Additionally, an analysis

of the memory access by threads should be done to determine if better utilization of the thread-local cache is possible, which may offset some of the overhead associated with parallelization. Lastly, an algorithm analysis could be done to further substantiate the empirical results of the sequential implementation analysis which exhibits large growth as the size of the data set increases and provide additional insights or suggest modifications to the algorithms. Many opportunities for algorithm optimization and modifications exist and should be explored.

References

[1] M. Riesenhuber and T. Poggio. Hierarchical models of object recognition in cortex. *Nature Neuroscience*, 2(11):1019–1025, November 1999.

[2] Y. LeCun and Y. Bengio. Convolutional networks for images, speech, and time-series. In M. A. Arbib, editor, *The Handbook of Brain Theory and Neural Networks*. MIT Press, 1995.

[3] G E Hinton. Learning multiple layers of representation. *Trends in Cognitive Sciences*, 11(10):428–34, 2007.

[4] J. Johnston. *The Allure of Machinic Life: Cybernetics, Artificial Life, and the New AI*. MIT Press, Cambridge, MA, 2008.

[5] G. Klir. *Facets of Systems Science*. Kluwer Academic, NY, 2nd edition, 2001.

[6] G. Lendaris. On systemsness and the problem solver: Tutorial comments. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-16(4):604–610, July/August 1986.

[7] J. Hawkins, S. Ahmad, and D. Dubinsky. Hierarchical temporal memory including htm cortical learning algorithms. Technical report, Numenta, CA, 2010.

[8] J. Hawkins and S. Blakeslee. *On Intelligence*. Times Books, New York, NY, 2004.

[9] D. George. *How the Brain Might Work: A Hierarchical and Temporal Model for Learning and Recognition*. PhD thesis, Stanford University, 2008.

[10] D. George and J. Hawkins. Towards a mathematical theory of cortical micro-circuits. *PLoS Comput Biol*, 5(10):e1000532, 10 2009.

[11] S. Garalevicius. Memory-prediction framework for pattern recognition: Performance and suitability of the bayesian model of visual cortex. In *FLAIRS Conference*, pages 92–97, FL, 2007.

[12] J. Thornton, T. Gustafsson, M. Blumenstein, and T. Hine. Robust character recognition using a hierarchical bayesian network. In *AI 2006: Advances in Artificial Intelligence*, volume 4304 of *Lecture Notes in Computer Science*, pages 1259–1264. Springer Berlin / Heidelberg, 2006.

[13] I. Arel, D. Rose, and R. Coop. Destin: A scalable deep learning architecture with application to high-dimensional robust pattern recognition. In *Proc. AAAI 2009 Fall Symposium on Biologically Inspired Cognitive Architectures (BICA)*, November 2009.

[14] J. C. Chappelier and A. Grumbach. RST: a connectionist architecture to deal with spatiotemporal relationships. *Neural Computation*, 10:883–902, 1998.

[15] J. C. Chappelier, M. Gori, and A. Grumbach. Time in connectionist models. In R. Sun and C. Giles, editors, *Sequence Learning*, volume 1828 of *Lecture Notes in Computer Science*, pages 105–134. Springer Berlin / Heidelberg, 2001.

[16] W. Gropp, E. Lusk, and R. Thakur. *Using OpenMPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, Cambridge, MA, 1999.

[17] B. Chapman, G. Jost, and R. van der Pas. *Using OpenMP: portable shared memory parallel programming.* MIT Press, Cambridge, MA, 2008.

[18] Y. LeCun. THE MNIST DATABASE of handwritten digits. `http://yann.lecun.com/exdb/mnist/`. [Jan 2011].

[19] Intel Corporation. Intel VTune Amplifier XE Performance Profiler. `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`. [May 2011].

[20] D. van Heesch. Doxygen. `http://www.doxygen.org/`. [June 2011].

Appendix A

Software Guide

A.1 Introduction

This brief introduction to the software is intended to provide the user with enough know how to run the software from the source code. The topics covered in this appendix include required packages, compiling the source code, setting network parameters and reading data files. Detailed reference material for the source code, including collaboration diagrams for each class, can be found in Appendix B.

A.2 Requirements

Boost The Boost C++ libraries are required and can be found at www.boost.org. Our software makes limited use of these libraries (only for random number generation) and no specific version of the library is required. We're using version 1.40 on Linux and version 1.44 on Windows.

OpenMP To take advantage of the multi-threaded version using OpenMP directives, a compiler that implements the OpenMP API must be used. A list of many of the compilers that implement the OpenMP API can be found at http://openmp.org/wp/openmp-compilers/. We're using the GNU g++ compiler on Linux and Visual Studio 2008 with the Visual C++ compiler on Windows.

A.3 Getting Started

A.3.1 Compiling

A makefile is provided for compiling on Linux using the GNU g++ compiler. Make sure to compile with the -fopenmp option if you wish to run the software using multiple threads. Currently the number of threads is set in the source file level.cpp, not at runtime. After compiling, type 'htm' to run. We've used Visual Studio 2008 for development on Windows and recommend Visual Studio for compiling and running the source code on Windows since Visual C++ supports OpenMP. Build files are not provided for Visual Studio, so the source files should be added as a new project and built "from scratch". Alternatively, you can compile the source code the Linux way using Cygwin or MinGW on Windows but we have not tested this.

A.3.2 Setting Network Parameters

A number of network parameters are available for the user to set in the source file topology.cpp. Information about these parameters is provided in Appendix B. The verification tests can be run using the macros in the source file topology.h. Uncommenting *VERIFY*, *TEST_2*, *TEST_3_4*, *HIGHER_O*, and *HIGHER_O2A* in topology.h will enable the verification tests described in sections 4.2 and 4.3.

A.3.3 Datasets

The Example class, implemented in the source file example.cpp, provides a container and functions for reading in an example from a file and storing it in a temporary object that is passed to the network. The user must specify the relative paths to the training and test data. This is done in the source file example.cpp by specifying the string variables *trainfiles* and *testfiles*. Additionally, the number of files to be used can be set with the *NUM_FILES* variable. Input data is expected to be presented with a newline separating individual input patterns and a ';' (preceded and followed by newlines) separating input sequences. Of course, the user is encouraged to modify the source file example.cpp or write their own interface to handle the desired data format or their choice.

A.3.4 Network Output

Network output is written to "output.txt", a file created at runtime. Typically, the input pattern and level output are written at each time step. The source code can be modified to write only the level output at each time step, if say, the user desires to pass the network's output to a classifer such as SVM of kNN.

Appendix B

Software Reference

This software reference guide contains descriptions of nearly all the member functions and attributes of each class. It is designed to provide the reader with a better understanding of how the implementation was designed. Collaboration diagrams are shown for each class. In these diagrams, objects are represented as boxes and dotted lines indicate a reference or pointer to another documented class, with the corresponding reference or pointer name given beside the dotted line. This reference documentation and was created using Doxygen [20], a software package designed for rapid generation of detailed documentation.

B.1 data Namespace Reference

Variables

- static std::string **trainfiles** []

  Relative paths and filenames for training data.

- static std::string **testfiles** []

  Relative paths and filenames for training data.

- const int **NUM_FILES** = 10

The number of files to use in the run.

B.1.1 Variable Documentation

B.1.1.1 std::string data::testfiles[] `[static]`

Initial value:

```
 {
"data_files/test/5/sequence1.test",
"data_files/test/5/sequence2.test",
"data_files/test/5/sequence3.test",
"data_files/test/5/sequence4.test",
"data_files/test/5/sequence5.test",
"data_files/test/5/sequence6.test",
"data_files/test/5/sequence7.test",
"data_files/test/5/sequence8.test",
"data_files/test/5/sequence9.test",
"data_files/test/5/sequence10.test"}
```

Relative paths and filenames for training data.

B.1.1.2 std::string data::trainfiles[] `[static]`

Initial value:

```
 {
"data_files/train/20/sequence1.train",
"data_files/train/20/sequence2.train",
```

```
"data_files/train/20/sequence3.train",
"data_files/train/20/sequence4.train",
"data_files/train/20/sequence5.train",
"data_files/train/20/sequence6.train",
"data_files/train/20/sequence7.train",
"data_files/train/20/sequence8.train",
"data_files/train/20/sequence9.train",
"data_files/train/20/sequence10.train"}
```

Relative paths and filenames for training data.

B.2 Cell Class Reference

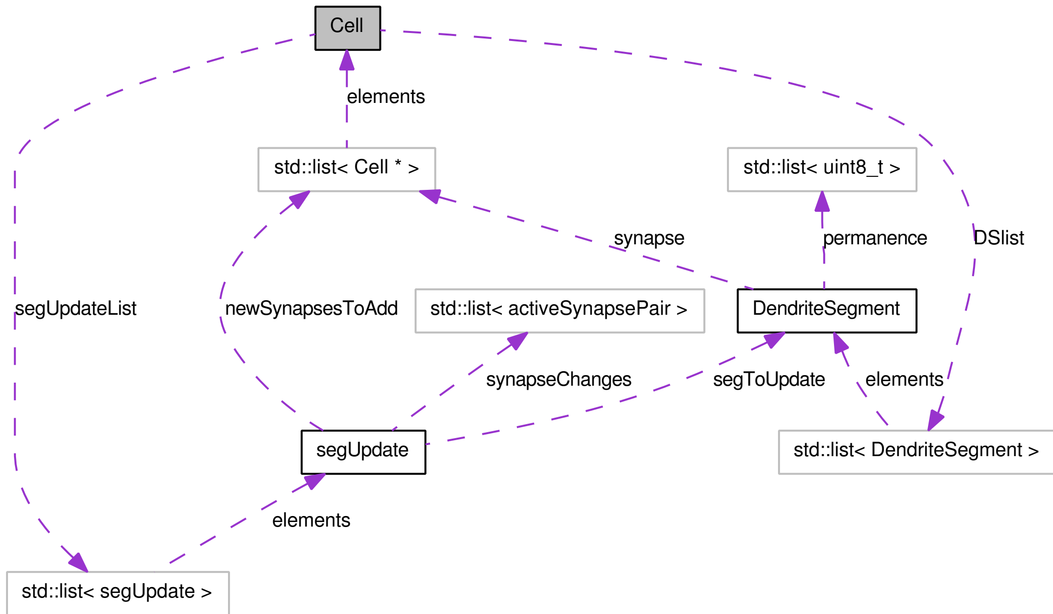

Figure B.1: Collaboration diagram for Cell.

Public Member Functions

- bool **setCell** (const unsigned int &numLatNbrCols)

Friends

- class **TemporalPooler**

B.2.1 Member Function Documentation

B.2.1.1 bool Cell::setCell (const unsigned int & *numLatNbr*)

The setCell routine determines a cell's lateral connectivity with its neighboring cells. The possibility of a connection is based on a random distribution. Or, the cell may be allowed to form connections with all lateral cells within its learning radius. This is done during the verification process, for example.

B.3 Column Class Reference

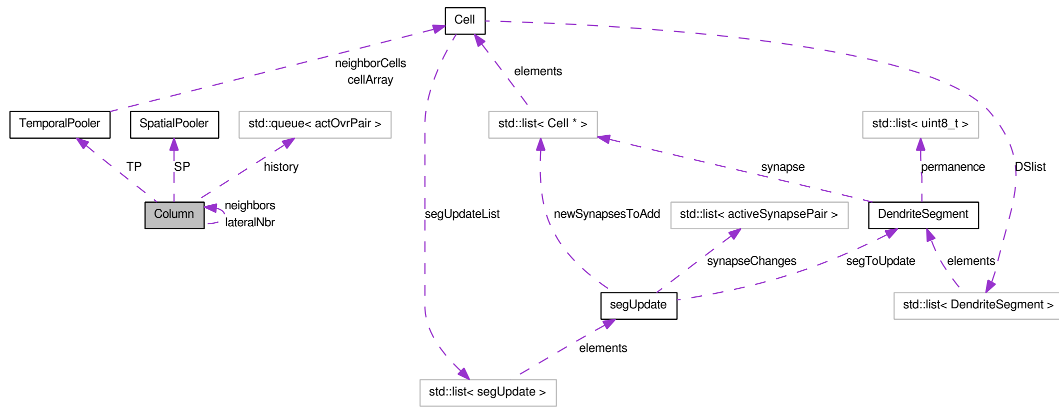

Figure B.2: Collaboration diagram for Column.

Public Member Functions

- bool **setSP** (**XYindex** &In_data_upper_left_xy)
- bool **setTP** ()
- bool **setNeighbors** (int row, int col, **Column** ∗∗&columnArray)
- bool **setCellNeighbors** ()
- **Cell** ∗ **getCells** (int &cellIndex)

  Returns a pointer to specific cell in a column.

- bool **computeSPScore** (std::vector< bool > &dataInl)
- bool **isActive** ()
- bool **SPUpdateActiveColPerm** ()

Calls the Spatial Pooler member function UpdateActiveColPerm to increment synapses connected to active input bits and decrement ones connected to inactive bits.

- bool **SPColumnLearning** (float &numPresentations)
- bool **updateDutyCycle** (float &numPresentations)
- float **maxDutyCycleNeighbors** ()

  Returns the highest active duty cycle from a column's neighbors.

- unsigned int **getScore** ()

  Returns a column's overlap score.

- float **getDutyCycle** ()

  Returns a column's active duty cycle.

- bool **TPphase1** ()

  Calls the Temporal Pooler member function phase1.

- bool **TPphase1start** ()

  Calls the Temporal Pooler member function phase1start.

- bool **TPphase2** ()

  Calls the Temporal Pooler member function phase2.

- bool **TPphase3** ()

  Calls the Temporal Pooler member function phase3.

- bool **infComputeScore** (std::vector< bool > &test_data)
- bool **infIsActive** ()
- bool **TPinfPhase1** ()

  Calls the Temporal Pooler member function infPhase1.

- bool **TPinfPhase2** ()

  Calls the Temporal Pooler member function infPhase2.

- bool **ffOutput** (std::vector< bool * > &levelOutput)

  Calls the Temporal Pooler member function ffOutput to generate feedforward output.

- bool **updateStates** ()

  Calls the Temporal Pooler member function updateStates.

- bool **clearStates** ()

  Calls the Temporal Pooler member function clearStates.

- bool **printStates** ()

  Calls the Temporal Pooler member function printStates.

- bool **printSS** ()

    Calls the Temporal Pooler member function printSS.

B.3.1 Member Function Documentation

B.3.1.1 bool Column::computeSPScore (std::vector< bool > & *dataIn*)

Calls the Spatial Pooler member function computeMatch to compute the column's overlap score with the current input.

Parameters

dataIn Input data for the current time step.

B.3.1.2 bool Column::infComputeScore (std::vector< bool > & *test_data*)

Calls the inference only version of the Spatial Pooler member function computeMatch which calculates a column's overlap score with the current input. No learning is performed.

Parameters

test_data The input data for inference only mode.

B.3.1.3 bool Column::infIsActive ()

Inference only version of the isActive function. The overlap score of a column is compared to its neighbors. If its score is greater than the nth score, where n is determined by the parameter desiredLocalActivity, the function returns true and the column is added to the list of active columns. If the score is less than then nth score, then the column is inhibited and the function returns false. This process is not done during verification testing. No learning is performed.

B.3.1.4 bool Column::isActive ()

The overlap score of a column is compared to its neighbors. If its score is greater than the nth score, where n is determined by the parameter desiredLocalActivity, the function returns true and the column is added to the list of active columns. If the score is less than then nth score, then the column is inhibited and the function returns false. This process is not done during verification testing.

B.3.1.5 bool Column::setCellNeighbors ()

Calls the Temporal Pooler member function setLateralNeighbors to create pointers to a column's neighboring cells.

Parameters

lateralNbr A column's array of pointers to neighboring columns.

numLatNbr The number of lateral neighbors this column has.

B.3.1.6 bool Column::setNeighbors (int *rowPosition*, int *colPosition*, Column **& *columnArray*)

Sets a column's neighborhoods for spatial and temporal pooling based on the column's xy position in the column array. Pointers to neighboring columns are created using the column's position and the inhibition radius and learning radius parameters. A special configuration is made for the verification tests.

Parameters

rowPosition This column's row position in the column array.

colPosition This column's column position in the column array.

columnArray The array of columns at a level in the network.

B.3.1.7 bool Column::setSP (XYindex & *SPIn_data_upper_left_xy*)

Initializes a column's spatial pooling object by calling the Spatial Pooler constructor.

Parameters

SPIn_data_upper_left_xy Determined by the level class, this parameter maps a column to specific subset of input space.

B.3.1.8 bool Column::setTP ()

Initializes a column's temporal pooler object by calling the constructor of the temporal pooling class.

Parameters

numLatNbr The number of lateral neighbors this column has.

B.3.1.9 bool Column::SPColumnLearning (float & *numPresentations*)

Calculates the minimum duty cycle based on the maximum duty cycle value of a column's neighbors. If the column's active duty cycle is less than the minimum, then the Spatial Pooler member function increaseBoost is called to increase the column's boost value. If the column's overlap duty cycle is less than the minimum, then Spatial Pooler member function increasePermanences is called to increment all of the column's proximal dendrite synapses.

Parameters

numPresentations The number of input patterns that have been presented thus far.

B.3.1.10 bool Column::updateDutyCycle (float & *numPresentations*)

Done before the column member function SPColumnLearning. This routine updates a column's active duty cycle. If the number of presentations is less than the average duty window, then the rolling average is used to calculate the active duty cycle and overlap duty cycle. If the number of presentations is greater than the average duty window, then the moving average is used and the first item in the column's history queue is processed.

Parameters

numPresentations The number of input patterns that have been presented thus far.

B.4 DendriteSegment Class Reference

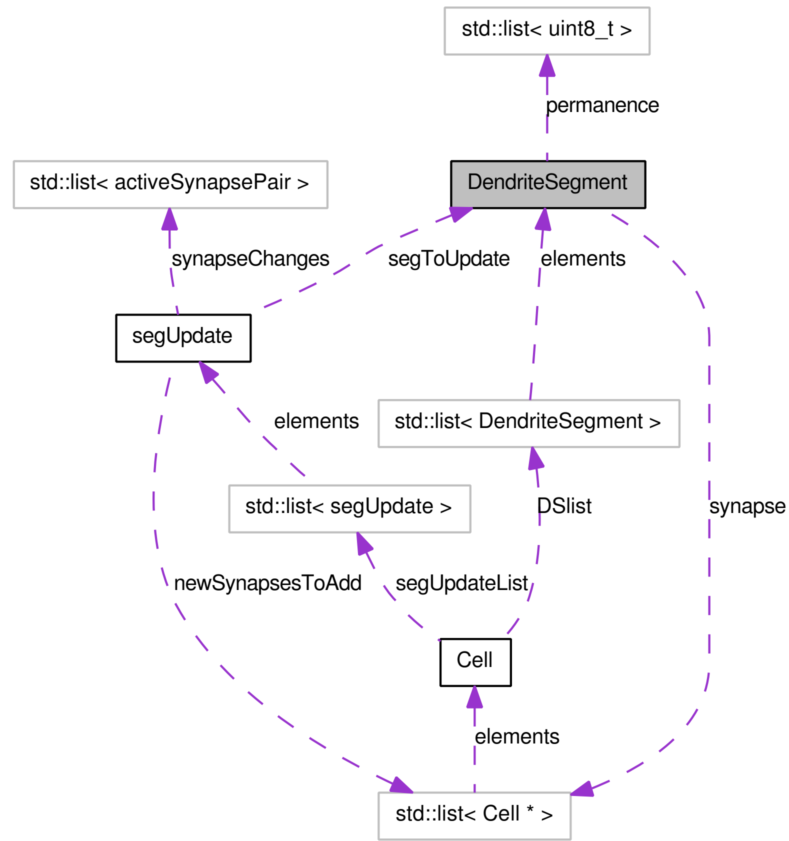

Figure B.3: Collaboration diagram for DendriteSegment.

Friends

- class **TemporalPooler**

B.5 Example Struct Reference

Public Member Functions

- bool **open_file** (std::string &filename)
- bool **close_file** ()
- bool **read_next** (**Network** *htmNetwork, bool &new_seq)

Public Attributes

- std::vector< bool > **stimulus**

  Container for the current input pattern.

- std::fstream **data**

  Filestream object for reading from current data file.

B.5.1 Detailed Description

Provides a means for reading in an example from a data file that is formatted with a newline delimiting input patterns and SEQ_DELIM delimiting input sequences. A filestream is opened, the next input pattern is read and stored in the public attribute ‘stimulus’. A member function is provided to close the filestream after all the input sequences are read.

B.5.2 Member Function Documentation

B.5.2.1 bool Example::read_next (Network * *htmNetwork*, bool & *new_seq*) `[inline]`

Checks that the file has opened before reading the next input. If the next character is the sequence delimiter, then the network states are cleared by calling the Network member function clearStates, and the the new_seq flag is set to true. If the end of the file has not been reached, then the next input pattern is read and parsed into input bits which are stored in the Example public attribute 'stimulus'.

Parameters

htmNetwork A pointer to the network.

new_seq A bool flag set to true at the beginning of a new input sequence.

B.6 Level Class Reference

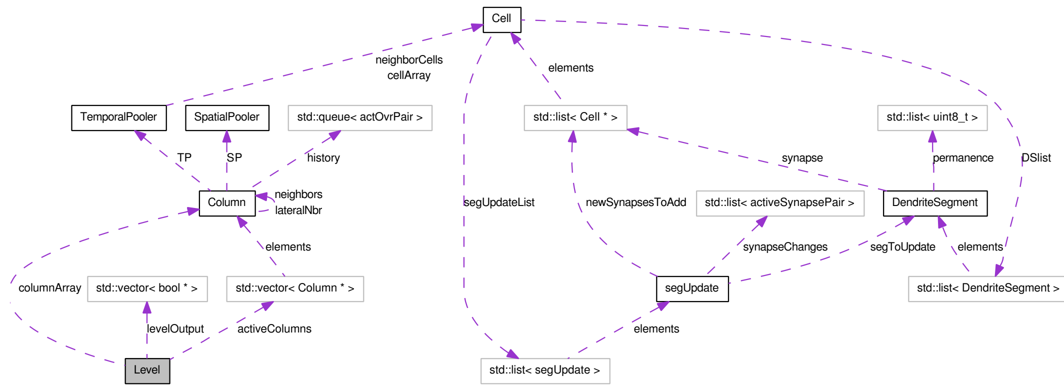

Figure B.4: Collaboration diagram for Level.

Public Member Functions

- **Level** ()
- bool **SPoverlap** (std::vector< bool > &dataIn)
- bool **SPinhibition** ()
- bool **SPlearning** (float &numPresentations)
- bool **TPlearning** ()
- bool **TPlearn1st** ()
- bool **SPinference** (std::vector< bool > &test_data)
- bool **TPinference** ()
- bool **generateOutput** ()
- bool **updateStates** ()
- bool **clearStates** ()

- bool **printStates** ()
- void **generateData** (std::vector< std::vector< int > > &data)
- void **generateData** (std::vector< std::vector< int > > &data1, std::vector< std::vector< int > > &data2)
- bool **Verify2** (std::vector< std::vector< int > > *data_ptr, float &numPresentations)
- bool **printSS** ()

B.6.1 Constructor & Destructor Documentation

B.6.1.1 Level::Level ()

The constructor for the Level class creates the array of columns and then determines the mapping of each column to the input space by calculating a pair of coordinates which correspond to the upper left corner of a column's receptive field. The constructor then calls the Column class member functions setSP, setNeighbors, setTP and setCellNeigbors to create a Spatial Pooler object, set each column's neighborhood, create a Temporal Pooler object and cells, and then set each cell's lateral connections.

B.6.2 Member Function Documentation

B.6.2.1 bool Level::clearStates ()

Calls the Column member function clearStates for every column in the level. The level's list of active columns is also cleared.

B.6.2.2 void Level::generateData (std::vector< std::vector< int > > & *data*, std::vector< std::vector< int > > & *data2*)

Generates data containing shared subsequences for the verification tests.

Parameters

> ***data*** A 2d vector container for storing the verification data.
>
> ***data2*** Another 2d vector container for storing the verification data.

B.6.2.3 void Level::generateData (std::vector< std::vector< int > > & *data*)

Generates the data for the verification tests.

Parameters

> ***data*** A 2d vector container for storing the verification data.

B.6.2.4 bool Level::generateOutput ()

Calls the Column member function ffOutput to generate the feedforward output for a level, then prints the output to stdout for every column in the level.

B.6.2.5 bool Level::printSS ()

Prints a column's row and column position and calls the Column member function printSS for every column in the level. Mainly used for the verification tests.

B.6.2.6 bool Level::printStates ()

Prints a column's row and column position and calls the Column member function printStates for every column in the level. Mainly used for debugging.

B.6.2.7 bool Level::SPinference (std::vector< bool > & *test_data*)

Executes the spatial pooling functions associated with inference. The Column member functions infComputeScore and infIsActive are called for all columns in the level.

Parameters

test_data Input data for inference only.

B.6.2.8 bool Level::SPinhibition ()

Executes the local inhibition step of spatial pooling by calling the Column member function isActive for every column in the level. Active columns are added to the level's list of active columns.

B.6.2.9 bool Level::SPlearning (float & *numPresentations*)

Executes the spatial pooling functions associated with learning. The Column member functions updateDutyCycle, SPUpdateActiveColPerm, and SPColumnLearning are called. updateDutyCycle and SPColumnLearning are called for all columns in the level and SPUpdateActiveColPerm is called for all currently active columns.

B.6.2.10 bool Level::SPoverlap (std::vector< bool > & *dataIn*)

Computes the overlap score of each column by calling the Column member function computeSPScore for every column in the level.

Parameters

> ***dataIn*** The input data for the current time step.

B.6.2.11 bool Level::TPinference ()

Executes the temporal pooling functions associated with inference. The Column member functions TPinfPhase1, and TPinfPhase2 are called. TPinfPhase2 is called for all columns in the level and TPinfPhase1 is called for all currently active columns.

B.6.2.12 bool Level::TPlearn1st ()

Executes the temporal pooling functions associated with learning when the first input pattern in a new sequence is presented. The Column member functions TPphase1start, TPphase2, and TPphase3 are called. TPphase2 and TPphase3 are called for all columns in the level and TPphase1start is called for all currently active columns.

B.6.2.13 bool Level::TPlearning ()

Executes the temporal pooling functions associated with learning. The Column member functions TPphase1, TPphase2, and TPphase3 are called. TPphase2 and

TPphase3 are called for all columns in the level and TPphase1 is called for all currently active columns.

B.6.2.14 bool Level::updateStates ()

Calls the Column member function updateStates for every column in the level. The level's list of active columns is also cleared.

B.6.2.15 bool Level::Verify2 (std::vector< std::vector< int > > * *data_ptr*, float & *numPresentations*)

Compares the predictions of each column to the next input to determine if a verification test is passed or failed. Results are printed to stdout.

Parameters

data_ptr A pointer to the 2d vector of verification data.

numPresentations The number of input presentations thus far.

B.7 Network Class Reference

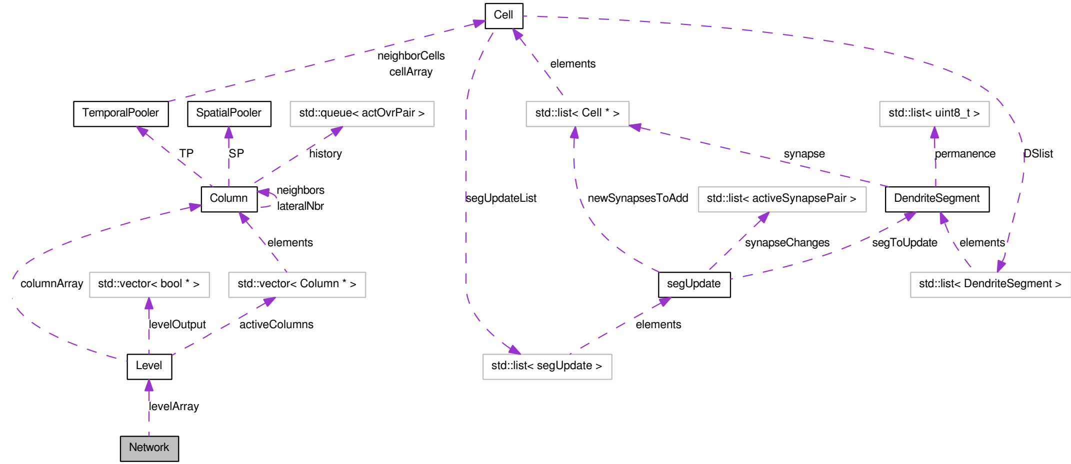

Figure B.5: Collaboration diagram for Network.

Public Member Functions

- bool **runSPoverlap** (std::vector< bool > &dataIn)

  Increments the number of presentations (numPresentations) and calls the Level member function SPoverlap.

- bool **runSPinhibition** ()

  Calls the Level member function SPinhibition.

- bool **runSPlearning** ()

  Calls the Level member function SPlearning.

- bool **runTPlearning** ()

Calls the Level member function TPlearning.

- bool **runTPlearn1st** ()

  Calls the Level member function TPlearn1st.

- bool **inference** (std::vector< bool > &test_data)
- bool **levelOutput** ()

  Calls the Level member function generateOutput.

- bool **updateStates** ()

  Calls the Level member function updateStates.

- bool **clearStates** ()

  Calls the Level member function clearStates and resets the number of presentations.

- bool **levelOutput** (std::vector< std::vector< int > > *data_ptr)

  Used for verification testing. Calls the Level member function generateOutput and Verify2.

- void **genData** (std::vector< std::vector< int > > &data)

  Used for verification testing. Calls the Level member function generateData.

- void **genData** (std::vector< std::vector< int > > &data1, std::vector< std::vector< int > > &data2)

  Used for verification testing. Calls the Level member function generateData to generate data with shared subsequences.

- bool **printSS** ()

  Mainly used for verification testing. Calls the Level member function printSS.

Public Attributes

- float **numPresentations**

B.7.1 Member Function Documentation

B.7.1.1 bool Network::inference (std::vector< bool > & *test_data*)

Increments the number of presentations (numPresentations) and calls the Level member functions associated with inference, SPinference and TPinference.

Parameters

test_data Input data for inference.

B.8 segUpdate Struct Reference

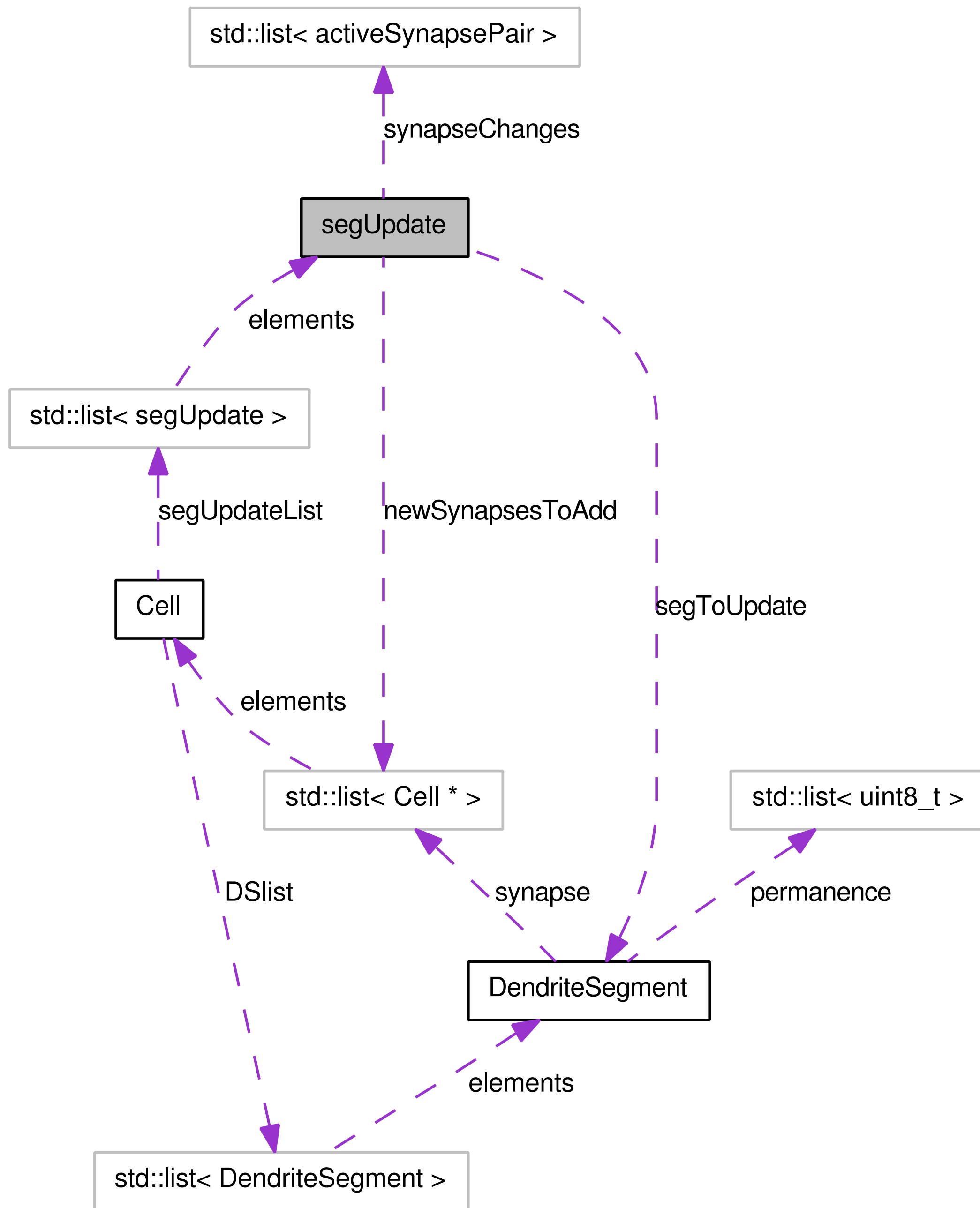

Figure B.6: Collaboration diagram for segUpdate.

Public Attributes

- **DendriteSegment * segToUpdate**
- std::list< **activeSynapsePair** > **synapseChanges**
- std::list< **Cell** * > **newSynapsesToAdd**
- bool **sequenceSegment**

B.8.1 Member Data Documentation

B.8.1.1 std::list<Cell *> segUpdate::newSynapsesToAdd

Pointers to cells which will be added to a dendrite Segments's synapse list with permenance equal to initialPerm.

B.8.1.2 DendriteSegment* segUpdate::segToUpdate

A pointer (initially null) to the segment to be updated.

B.8.1.3 bool segUpdate::sequenceSegment

A bool (initially false) to indicate if the segment is a sequence segment

B.9 SpatialPooler Class Reference

Public Member Functions

- **SpatialPooler** (**XYindex** &In_data_upper_left_xy)
- unsigned int **computeMatch** (std::vector< bool > &dataIn, std::queue< **actOvrPair** > &history)
- bool **UpdateActiveColPerm** ()
- bool **increaseBoost** (float &dutyCycleDifference)
- bool **increasePermanences** ()
- unsigned int **computeMatch** (std::vector< bool > &dataIn)

B.9.1 Constructor & Destructor Documentation

B.9.1.1 SpatialPooler::SpatialPooler (XYindex & *In_data_upper_left_xy*)

The constructor takes an STL Pair argument which indicates xy coordinates in the input space to which the upper left corner of a this column's receptive field will be mapped. Arrays of potential synapses and permenances are created initalized using a random distribution function.

Parameters

In_data_upper_left_xy An stl pair that maps to the input space.

B.9.2 Member Function Documentation

B.9.2.1 unsigned int SpatialPooler::computeMatch (std::vector< bool > & *dataIn*)

Run during inference only mode. Computes the number of valid synapses that are aligned with active input at the current time step. If an input bit is active and the associated synapses have a permenance value above threshold, then the score is incremented. If the resulting score is above the threshold for minimum overlap (minOverlap), then the score is multiplied by the column's boost value. Otherwise, if the score is below minThreshold, the score is set to 0. No overlap history is updated.

Parameters

> ***dataIn*** The input data.

Returns

> A column's boosted score is returned.

B.9.2.2 unsigned int SpatialPooler::computeMatch (std::vector< bool > & *dataIn*, std::queue< actOvrPair > & *history*)

Computes the number of valid synapses that are aligned with active input at the current time step. If an input bit is active and the associated synapses have a permenance value above threshold, then the score is incremented. If the resulting score is above the threshold for minimum overlap (minOverlap), then the score is

multiplied by the column's boost value and a column's history is updated to reflect that the input resulted in significant overlap at this time step. Otherwise, if the score is below minThreshold, the score is set to 0 and the history is updated to reflect that the input did not result in significant overlap at this time step.

Parameters

dataIn The input data.

history Keeps track of how often a column has had a significant match score for computing the overlap duty cycle.

Returns

A column's boosted score is returned.

B.9.2.3 bool SpatialPooler::increaseBoost (float & *dutyCycleDifference*)

Increases a column's boost value based on its duty cycle difference.

Parameters

dutyCycleDifference The difference between the minimum duty cycle and the column's active duty cycle.

B.9.2.4 bool SpatialPooler::increasePermanences ()

Increases the permanence values of all the synapses in a coulmn's proximal dendrite. This routine is called when a column's overlap duty cycle falls below the minimum duty cycle.

B.9.2.5 bool SpatialPooler::UpdateActiveColPerm ()

Updates the permenance of potential synapses that are aligned with active input at the current time step and decrements the permanences of synapses aligned with currently inactive input. Permenance values will not exceed or fall below the maximum and minimum values of 100 and 0.

B.10 TemporalPooler Class Reference

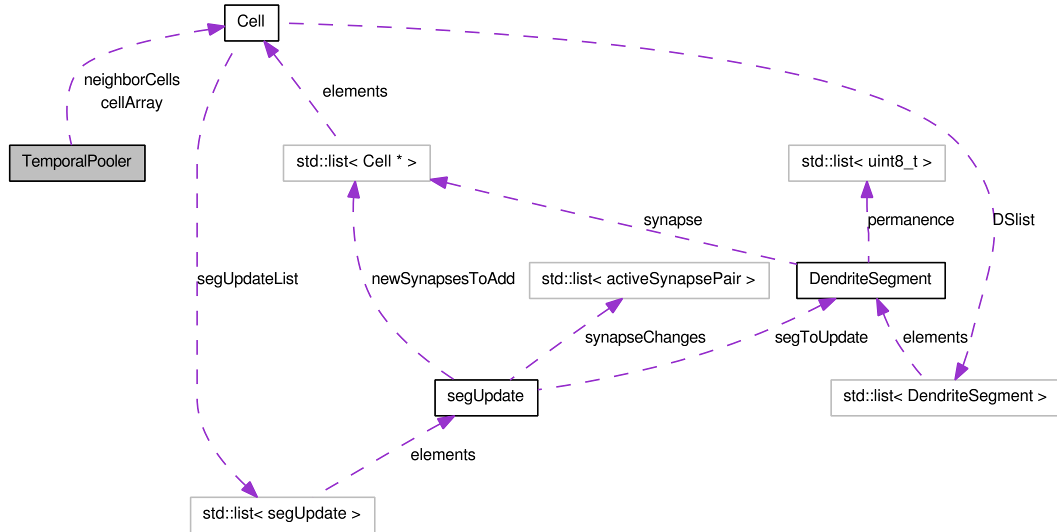

Figure B.7: Collaboration diagram for TemporalPooler.

Public Member Functions

- **TemporalPooler** (const unsigned int &numNeighborCells)
- bool **setLateralNeighbors** (**Column** **&colNeighbors, const unsigned int &numLatNbr)
- **Cell** * **getCells** (int &cellIndex)
- bool **segmentActive** (**DendriteSegment** *seg, const int timeStep, const int cellState)
- **DendriteSegment** * **getActiveSegment** (**Cell** *a_cell, const int timeStep, const int cellState)
- **DendriteSegment** * **getBestMatchingSegment** (**Cell** *currentCell, int &myMaxNumActiveSynapses)

- **Cell** * **getBestMatchingCell** (**DendriteSegment** *&selectedSeg)
- bool **getSegmentActiveSynapses** (**Cell** *bestCell, **segUpdate** &sUpdate, **DendriteSegment** *selectedSeg, const int &timeStep, const unsigned int &numNeighbors, bool newSynapses)
- bool **adaptSegments** (**Cell** *&adaptingCell, std::list< **segUpdate** > &segUpdateList, bool posReinforce)
- bool **phase1** (const unsigned int &numNeighbors)
- bool **phase1start** ()
- bool **phase2** (const unsigned int &numNeighbors)
- bool **phase3** ()
- bool **ffOutput** (std::vector< bool * > &levelOutput)
- bool **infPhase1** ()
- bool **infPhase2** ()
- bool **updateStates** ()
- bool **clearStates** ()
- bool **printStates** ()
- bool **printSS** ()

B.10.1 Constructor & Destructor Documentation

B.10.1.1 TemporalPooler::TemporalPooler (const unsigned int & *numLatNbr*)

The constructor of the Temporal Pooler class creates an array of cells and determines the lateral connections to neighboring cells.

Parameters

numLatNbr The number of lateral neighbors.

B.10.2 Member Function Documentation

B.10.2.1 bool TemporalPooler::adaptSegments (Cell *& *cellToAdapt*, std::list< segUpdate > & *segUpdateList*, bool *posReinforce*)

This routine iterates through a cell's list of segUpdates and reinforces each segment according the the posReinforce agrument. If posReinforce is True, then active synapses in the activeSynapsePair list get ther permanences incremented by permanenceInc and all other synapses get their permancences decremented by permanenceDec. If posReinforce is False, then active synapses in the activeSynapsePair list get their permanences decremented by permanenceDec. If the synToAdd list is not empty, new synapses are added with permanence equal to intialPerm. If the segment update points to NULL meaning the segment doesn't exist yet, a new segment is created and the synapses are added with initalPerm.

Parameters

cellToAdapt A pointer to the cell being updated.

segUpdateList The list of segUpdates to be adapted.

posReinforce An bool argument which indicates whether or not synpases will be incremented or decremented.

B.10.2.2 bool TemporalPooler::clearStates ()

Clears all states in a cell's state machine by setting all values of the current and previous states to false.

B.10.2.3 bool TemporalPooler::ffOutput (std::vector< bool * > & *levelOutput*)

Determines the output of every cell in the level. If the cell is either active or predictive, then a pointer to one of those two cell states is added to levelOutput, a vector of bool pointers, which represents the feedfoward output of the level. If the cell is neither active or predictive, a pointer to the cell's active state (which is false) is added to the levelOutput. The routine behaves slightly differently when it is run during the verification process. In this case, only pointers to the predictive state are added in order to verify that the appropriate predictions are being made based on the level output.

B.10.2.4 DendriteSegment * TemporalPooler::getActiveSegment (Cell * *cel*, const int *timeStep*, const int *cellState*)

Finds a dendrite segment stored by the given cell such that the routine 'segmentActive' is true. Preference is given to dendrite segments that are 'sequence segments' (those with the seqSeg flag set to True) with the most activity. If no active segments are found than a null pointer is returned.

Parameters

cel Points to the cell with the dendrite segment to evaluate.

timeStep The specified time step for the segmentActive routine.

cellState The specified cell state for the segmentActive routine.

Returns

A pointer to the selected active segment or null if none exists.

B.10.2.5 Cell ∗ TemporalPooler::getBestMatchingCell (DendriteSegment ∗& *selectedSeg*)

Finds the cell with the best matching segment by calling the getBestMatchingSegment routine for each cell in the column and comparing the maxNumActiveSynapses argument in order to find the best match. If no cell has a matching segment than the cell with the fewest number of segments is returned

Parameters

selectedSeg A DendriteSegment pointer that will point to a cell's best matching segment.

Returns

The best matching cell or cell with the minimum number of segments.

B.10.2.6 DendriteSegment * TemporalPooler::getBestMatchingSegment (Cell * *cel*, int & *myMaxNumActiveSynapses*)

For a given cell, finds the dendrite segment with the largest number of active synapses at the previous time step. The permanence value of the synapses is allowed to be below the connection threshold (TPthreshold) and the number of active synapses is allowed to be below activationThreshold but must be above the minimum threshold (minThreshold).

Parameters

cel A pointer to the specified cell.

myMaxNumActiveSynapses An interger argument for keeping track of the largest number of active synapses.

Returns

A pointer to the best matching dendrite segment or NULL if no segments are found.

B.10.2.7 Cell * TemporalPooler::getCells (int & *cellIndex*)

Returns a pointer to a specific cell in a column.

Parameters

cellIndex Specifies which cell to return a pointer to.

B.10.2.8 bool TemporalPooler::getSegmentActiveSynapses (Cell * *bestCell*, segUpdate & *sUpdate*, DendriteSegment * *selectedSeg*, const int & *timeStep*, const unsigned int & *numNeighbors*, bool *newSynapses*)

This is one of the most complex routines of the algorithm. It generates a dendrite segment update structure containing a list of proposed changes that will be made permanent with the adaptSegments routine. These changes may include making a list of pointers to existing synapses that should be reinforced, adding new synapses to an existing segment, or adding a new dendrite segment to the specified cell. If the segment to be updated already exists, then a list of active synapses is made. newSynapseCount - numActiveSynapses new synapses are added (if possible), if the optional argument newSynapses is true. If the argument selectedSeg is NULL, meaning the dendrite segment doesn't exist yet, then a segment update structure is created with newSynapseCount new synapses (if possible) with permanence equal to initialPerm.

Parameters

bestCell A pointer to the best cell returned by the getBestMatchingCell routine.

sUpdate A new segment update structure which will contain the proposed changes.

selectedSeg A pointer to the dendrite segment to be updated. Points to NULL if the segment doesn't exist yet.

timeStep The time step for determining which synapses are connected to

active cells.

numNeighbors The number of lateral neighbors best cell has.

newSynapses An optional bool argument. Set to true if new synapses are to be added.

Returns

True if a segment update was created successfully. Return false if there weren't any connected learning cells from previous time step and a new segment couldn't be created.

B.10.2.9 bool TemporalPooler::infPhase1 ()

No learning is done during the inference only version of Phase 1 of temporal pooling. Predictive cells from the previous time step are checked to see if the prediction is due to a dendrite segment that is marked as a 'sequence segment', indicating that it is predicting the current feed-forward input. If the prediction is due to a sequence segment, the botUpPredicted flag is set to true. If the botUpPredicted flag is still false after each cell in the active column has been evaluated, then all cells are set to active. No cells are set to the learn state and no segment update structures are stored in the inference only mode of Phase 1.

B.10.2.10 bool TemporalPooler::infPhase2 ()

No learning is done during the inference only version of Phase 2 of temporal pooling. Phase 2 calculates the predictive state for all cells in the network. A cell

enters the predictive state if one of its dendrite segments has enough of its lateral connections are connected to currently active cells. If not, it remains inactive. No segment updates are stored during the Phase 2 inference only mode.

B.10.2.11 bool TemporalPooler::phase1 (const unsigned int & *numNeighbors*)

This routine is run for each active column starting with the second pattern in an input sequence. Predictive cells from the previous time step are checked to see if the prediction is due to a dendrite segment that is marked as a 'sequence segment', indicating that it is predicting the current feed-forward input. If the prediction is due to a sequence segment, the botUpPredicted flag is set to true and the seqment is checked to see if it is active due to cells in the learn state at the previous time step. If so, the learnCellChosen flag is set to true and the cell is set to the learn state. After each cell in the active column as been evaluated, if the botUpPredicted flag is still false, then all cells are set to active. If the learnCellChosen flag is still false, then the best matching cell is found and a segment update structure is created with the sequence segment flag set to true by calling the getSegmentActiveSynapses sub-routine. The best matching cell is also set to the learn state.

Parameters

numNeighbors The number of lateral neighbors is passed to the getSegmentActiveSynapses sub-routine.

B.10.2.12 bool TemporalPooler::phase1start ()

This routine is run for each active column when the first input of a new input sequence is presented. Because this is a new input sequence, there is no prior input and no cells will be in a predictive state, thus all cells in the active column are set to active. The first cell in the column is placed in the learn state.

B.10.2.13 bool TemporalPooler::phase2 (const unsigned int & *numNeighbors*)

This routine calculates the predictive state for all cells in the network. A cell will enter the predictive state if one of its dendrite segments has enough of its lateral connections are connected to currently active cells. If so, active dendrite segments will store a new segment update. Additionally, a dendrite segment that could have predicted this activiation pattern will be selected to store a segment update as well.

Parameters

numNeighbors The number of lateral neighbors is passed to the getSegmentActiveSynapses sub-routine.

B.10.2.14 bool TemporalPooler::phase3 ()

Every cell in the network is evaluated in this routine. Cells currently in the learn state will have their segment update lists implemented with 'postive reinforcement'. Cells that were in the predictive state at the previous time step but are

not predictive at the current time step, will have their segment update list implemented with 'negative reinforcement'. In both cases the segment update lists are cleared after being implemented.

B.10.2.15 bool TemporalPooler::printSS ()

Prints the number of dendrite segments learned by each cell in a column and the number of synapses in each segment. Which segments are 'sequence segments' may also be printed optionally. This routine is used in the verification process and for aiding in debugging.

B.10.2.16 bool TemporalPooler::printStates ()

Prints a cell's current and previous states, as well as the number of dendrite segments learned by the cell and number of synapses in each of its dendrite segments. This routine is mainly used for parameter tuning and debugging.

B.10.2.17 bool TemporalPooler::segmentActive (DendriteSegment * *seg*, const int *timeStep*, const int *cellState*)

For a dendrite segment, time step and cell state, this routine determines if the number of valid synapses connected to active cells is greater than the activation threshold, 'activationThreshold'.

Parameters

seg Pointer to the given dendrite segment.

timeStep Specified time step (previous or current).

cellState Specified cell state (predictive, active, learn).

Returns

True if above the activation threshold, false if below.

B.10.2.18 bool TemporalPooler::setLateralNeighbors (Column **& *colNeighbors*, const unsigned int & *numLatNbr*)

Creates an array of pointers to neighboring cells by first creating an array of cell pointers and points them to each cell in this column's neighboring columns.

Parameters

colNeighbors An array of pointers to this column's neighbors.

numLatNbr The number of lateral neighbors for this column.

B.10.2.19 bool TemporalPooler::updateStates ()

Implements a 'time step' in a cell's state machine. A cell's current and previous states are updated. The previous state is set to the values of the current state and the current state is cleared by setting the values to false.

B.11 example.cpp File Reference

Classes

- struct **Example**

Namespaces

- namespace **data**

Variables

- static std::string **data::trainfiles** [ ]

  Relative paths and filenames for training data.

- static std::string **data::testfiles** [ ]

  Relative paths and filenames for training data.

- const int **data::NUM_FILES** = 10

  The number of files to use in the run.

- const char **SEQ_DELIM** = ';'

  Delimeter used in the datafile to separate input sequences.

- const int **NUM_BITS** = 784

  Number of input bits.

B.12 htm.cpp File Reference

Functions

- int **main** (int argc, char **argv)

B.12.1 Detailed Description

Contains the main function. Input patterns are read in one at a time using an Example struct. Training is performed with all input sequences in the specified training data files before inference is performed with all input sequences in the specified test data files. The various verification tests, which are contained here, can be enabled by using the macros described in the topology.h source file. Output is written to "output.txt".

B.13 params.h File Reference

Variables

- float **PROB**

  Probablity to be used for Bernouli and Binomial distributions.

- int **columnsPerRow**

  The number of columns per row in a square array.

- uint8_t **SPRFSize**

  Receptive field of a column's spatial pooler.

- int **inhibitionRadius**

  Inhibition radius for spatial pooling.

- int **learningRadius**

  Learning radius for temporal pooling.

- unsigned int **inputDim**

  Input dimension is assumed to be square.

- uint8_t **initialPerm**

  The initial permanence value for newly added synapses.

- uint8_t **SPthreshold**

  The permanence threshold for spatial pooler synapses (proximal dendrite synapses).

- unsigned int **minOverlap**

  The minimum overlap score a column must have to compete in local competition, otherwise score is set to 0.

- uint8_t **desiredLocalActivity**

  The number of columns that will be winners after the local inhibition step.

- uint8_t **numCells**

  The number of cells per column.

- unsigned int **newSynapseCount**

  The number of lateral connections to cells to store in a dendrite segment.

- uint8_t **TPthreshold**

  The permanence threshold for temporal pooler synapses(lateral connections in dendrite segments).

- uint8_t **activationThreshold**

  The number of cells that must be active in a dendrite segment for the segment to be considered active.

- uint8_t **minThreshold**

  The minimum number of active synapses required for a segment to be considered as a match in the getBestMatchingSegment routine.

- uint8_t **permanenceInc**

  The amount to increment permanence by during learning.

- uint8_t **permanenceDec**

  The amount to decrement permanence by during learning.

- float **avgDutyWindow**

  The number of previous presentations over which the moving average is calculated when determining activeDutCycle and overlapDutyCycle.

- uint8_t **increasePerm**

  The amount to increase all of a column's proximal dendrite synapse permanences when a column is underperforming.

B.13.1 Detailed Description

This file contains the extern declarations for the network parameters. Network parameters can be set by the user in the source file topology.cpp.

B.14 rng.h File Reference

Typedefs

- typedef boost::mt11213b **gen_type**
- typedef boost::bernoulli_distribution **bern_dist**
- typedef boost::variate_generator< gen_type &, bern_dist > **bern_gen**
- typedef boost::binomial_distribution **binom_dist**

- typedef boost::variate_generator< gen_type &, binom_dist > **binom_gen**
- typedef boost::uniform_int **uni_int_dist**
- typedef boost::variate_generator< gen_type &, uni_int_dist > **uni_int_gen**

B.14.1 Detailed Description

This file contains the type definition for the various random distributions and random number generators used.

B.15 topology.h File Reference

Typedefs

- typedef std::pair< uint8_t, uint8_t > **XYindex**
- typedef std::pair< bool, bool > **actOvrPair**

B.15.1 Detailed Description

This file contains the macros (not shown) for enabling and disabling the various verification tests. A few typedefs are also found here.

B.15.2 Typedef Documentation

B.15.2.1 actOvrPair

A STL Pair of bools used for maintaining a column's activation and overlap history. The first value stores a bool indicating if the column was active, the second value stores a bool indicating the column had significant overlap.

B.15.2.2 XYindex

A STL Pair used for indexing the X and Y dimensions in each 2D array. The first element is the column index and the second element is the row index.

B.16 tp.h File Reference

Classes

- struct **segUpdate**
- class **TemporalPooler**
- class **Cell**
- class **DendriteSegment**

Typedefs

- typedef std::pair< uint8_t *, bool > **activeSynapsePair**

Variables

Definitions of the three cell states, active, predictive, and learn. Each is maintained for two time steps, previous and current.

- const int **ACTIVE** = 0
- const int **PREDICTIVE** = 1
- const int **LEARN** = 2
- const int **PREVIOUS** = 0
- const int **CURRENT** = 1

B.16.1 Typedef Documentation

B.16.1.1 activeSynapsePair

A STL "pair" structure used for segment updates. Contains a pointer to each of a dendrite segment's permanence values that are associated with a synapse and a bool indicating if the corresponding synapse is active (true) or not (false).