HIERARCHICAL TEMPORAL MEMORY

including

HTM Cortical Learning Algorithms

VERSION 0.1, NOVEMBER 9, 2010

Read This First!

This is an early draft of this document. There are several things missing that you should be aware of.

What IS in this document:

This document describes in detail new algorithms for learning and prediction developed by Numenta in 2010. The new algorithms are described in sufficient detail that a programmer can understand and implement them if desired. It starts with an introductory chapter. If you have been following Numenta and have read some of our past white papers, the material in the introductory chapter will be familiar. The other material is new.

What is NOT in this document:

There are several topics related to the implementation of these new algorithms that did not make it into this early draft.

- Although most aspects of the algorithms have been implemented and tested in software, none of the test results are currently included.

- There is no description of how the algorithms can be applied to practical problems. Missing is a description of how you would convert data from a sensor or database into a distributed representation suitable for the algorithms.

- The algorithms are capable of on-line learning. A few details needed to fully implement on-line learning in some rarer cases are not described.

- There are several planned chapters that are not yet included. They include chapters on the biological basis of the algorithms, a discussion of the properties of sparse distributed representations, and chapters on applications and examples.

We are making this document available in its current form because we think the algorithms will be of interest to others. The missing components of the document should not impede understanding and experimenting with the algorithms by motivated researchers. We will revise this document regularly to reflect our progress.

Table of Contents

Future chapters will include information on applications, examples, comparison to other models of machine learning, a bibliography, a chapter on the biological basis of HTM, and sparse distributed representations.

Preface

There are many things humans find easy to do that computers are currently unable to do. Tasks such as visual pattern recognition, understanding spoken language, recognizing and manipulating objects by touch, and navigating in a complex world are easy for humans. Yet despite decades of research, we have few viable algorithms for achieving human-like performance on a computer.

In humans, these capabilities are largely performed by the neocortex. Hierarchical Temporal Memory (HTM) is a technology modeled on how the neocortex performs these functions. HTM offers the promise of building machines that approach or exceed human level performance for many cognitive tasks.

This document describes HTM technology. Chapter 1 provides a broad overview of HTM, outlining the importance of hierarchical organization, sparse distributed representations, and learning time-based transitions. Chapter 2 describes the HTM cortical learning algorithms in detail. Chapters 3 and 4 provide pseudocode for the HTM learning algorithms divided in two parts called the spatial pooler and temporal pooler. After reading chapters 2 through 4, experienced software engineers should be able to reproduce and experiment with the algorithms. Hopefully, some readers will go further and extend our work.

Intended audience

This document is intended for a technically educated audience. While we don't assume prior knowledge of neuroscience, we do assume you can understand mathematical and computer science concepts. We've written this document such that it could be used as assigned reading in a class. Our primary imagined reader is a student in computer science or cognitive science, or a software developer who is interested in building artificial cognitive systems that work on the same principles as the human brain.

Non-technical readers can still benefit from certain parts of the document, particularly *Chapter 1: HTM Overview*.

Software release

It is our intention to release software based on the algorithms described in this document in mid-2011.

Relation to previous documents

Parts of HTM theory are described in the 2004 book *On Intelligence*, in white papers published by Numenta, and in peer reviewed papers written by Numenta employees. We don't assume you've read any of this prior material, much of which has been incorporated and updated in this volume. Note that the HTM learning algorithms described in Chapters 2-4 have not been previously published. The new algorithms replace our first generation algorithms, called Zeta 1. For a short time, we called the new algorithms "Fixed-density Distributed Representations", or "FDR", but we are no longer using this terminology. We call the new algorithms the HTM Cortical Learning Algorithms, or sometimes just the HTM Learning Algorithms.

We encourage you to read *On Intelligence*, written by Numenta co-founder Jeff Hawkins with Sandra Blakeslee. Although the book does not mention HTM by name, it provides an easy-to-read, non-technical explanation of HTM theory and the neuroscience behind it. At the time *On Intelligence* was written, we understood the basic principles underlying HTM but we didn't know how to implement those principles algorithmically. You can think of this document as continuing the work started in *On Intelligence*.

About Numenta

Numenta, Inc. (www.numenta.com) was formed in 2005 to develop HTM technology for both commercial and scientific use. To achieve this goal we are fully documenting our progress and discoveries. We also publish our software in a form that other people can use for both research and commercial development. We have structured our software to encourage the emergence of an independent, application developer community. Use of Numenta's software and intellectual property is free for research purposes. We will generate revenue by selling support, licensing software, and licensing intellectual property for commercial deployments. We always will seek to make our developer partners successful, as well as be successful ourselves.

Numenta is based in Redwood City, California. It is privately funded.

About the authors

This document is a collaborative effort by the employees of Numenta. The names of the principal authors for each section are listed in the revision history.

Revision history

November 9, 2010: first release of Preface, Chapters 1, 2, 3, 4. Authors include Jeff Hawkins, Subutai Ahmad, and Donna Dubinsky.

Chapter 1 – HTM Overview

Hierarchical Temporal Memory (HTM) is a machine learning technology that aims to capture the structural and algorithmic properties of the neocortex.

The neocortex is the seat of intelligent thought in the mammalian brain. High level vision, hearing, touch, movement, language, and planning are all performed by the neocortex. Given such a diverse suite of cognitive functions, you might expect the neocortex to implement an equally diverse suite of specialized neural algorithms. This is not the case. The neocortex displays a remarkably uniform pattern of neural circuitry. The biological evidence suggests that the neocortex implements a common set of algorithms to perform many different intelligence functions.

HTM provides a theoretical framework for understanding the neocortex and its many capabilities. To date we have implemented a small subset of this theoretical framework. Over time, more and more of the theory will be implemented. Today we believe we have implemented a sufficient subset of what the neocortex does to be of commercial and scientific value.

Programming HTMs is unlike programming traditional computers. With today's computers, programmers create specific programs to solve specific problems. By contrast, HTMs are trained through exposure to a stream of sensory data. The HTM's capabilities are determined largely by what it has been exposed to.

HTMs can be viewed as a type of neural network. By definition, any system that tries to model the architectural details of the neocortex is a neural network. However, on its own, the term "neural network" is not very useful because it has been applied to a large variety of systems. HTMs model neurons (called cells when referring to HTM), which are arranged in columns, in layers, in regions, and in a hierarchy. The details matter, and in this regard HTMs are a new form of neural network.

As the name implies, HTM is fundamentally a memory based system. HTM networks are trained on lots of time varying data, and rely on storing a large set of patterns and sequences. The way data is stored and accessed is logically different from the standard model used by programmers today. Classic computer memory has a flat organization and does not have an inherent notion of time. A programmer can implement any kind of data organization and structure on top of the flat computer memory. They have control over how and where information is stored. By contrast, HTM memory is more restrictive. HTM memory has a hierarchical organization and is inherently time based. Information is always stored in a distributed fashion. A user of an HTM specifies the size of the hierarchy and what to train the system on, but the HTM controls where and how information is stored.

Although HTM networks are substantially different than classic computing, we can use general purpose computers to model them as long as we incorporate the key functions of hierarchy, time and sparse distributed representations (described in detail later). We believe that over time, specialized hardware will be created to generate purpose-built HTM networks.

In this document, we often illustrate HTM properties and principles using examples drawn from human vision, touch, hearing, language, and behavior. Such examples are useful because they are intuitive and easily grasped. However, it is important to keep in mind that HTM capabilities are general. They can just as easily be exposed to non-human sensory input streams, such as radar and infrared, or to purely informational input streams such as financial market data, weather data, Web traffic patterns, or text. HTMs are learning and prediction machines that can be applied to many types of problems.

HTM principles

In this section, we cover some of the core principles of HTM: why hierarchical organization is important, how HTM regions are structured, why data is stored as sparse distributed representations, and why time-based information is critical.

Hierarchy

An HTM network consists of regions arranged in a hierarchy. The region is the main unit of memory and prediction in an HTM, and will be discussed in detail in the next section. Typically, each HTM region represents one level in the hierarchy. As you ascend the hierarchy there is always convergence, multiple elements in a child region converge onto an element in a parent region. However, due to feedback connections, information also diverges as you descend the hierarchy. (A "region" and a "level" are almost synonymous. We use the word "region" when describing the internal function of a region, whereas we use the word "level" when referring specifically to the role of the region within the hierarchy.)

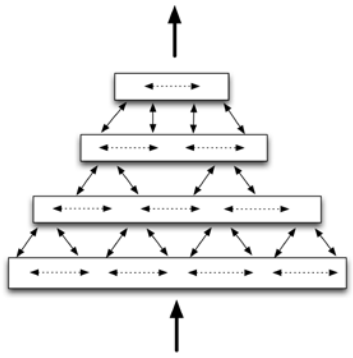

Figure 1.1: Simplified diagram of four HTM regions arranged in a four-level hierarchy, communicating information within levels, between levels, and to/from outside the hierarchy

It is possible to combine multiple HTM networks. This kind of structure makes sense if you have data from more than one source or sensor. For example, one network might be processing auditory information and another network might be processing visual information. There is convergence within each separate network, with the separate branches converging only towards the top.

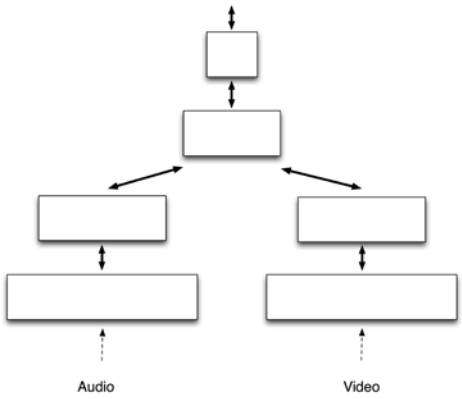

Figure 1.2: Converging networks from different sensors

The benefit of hierarchical organization is efficiency. It significantly reduces training time and memory usage because patterns learned at each level of the hierarchy are reused when combined in novel ways at higher levels. For an illustration, let's consider vision. At the lowest level of the hierarchy, your brain stores information about tiny sections of the visual field such as edges and corners. An edge is a fundamental component of many objects in the world. These low-level patterns are recombined at mid-levels into more complex components such as curves and textures. An arc can be the edge of an ear, the top of a steering wheel or the rim of a coffee cup. These mid-level patterns are further combined to represent high-level object features, such as heads, cars or houses. To learn a new high level object you don't have to relearn its components.

As another example, consider that when you learn a new word, you don't need to relearn letters, syllables, or phonemes.

Sharing representations in a hierarchy also leads to generalization of expected behavior. When you see a new animal, if you see a mouth and teeth you will predict that the animal eats with his mouth and that it might bite you. The hierarchy enables a new object in the world to inherit the known properties of its sub-components.

How much can a single level in an HTM hierarchy learn? Or put another way, how many levels in the hierarchy are necessary? There is a tradeoff between how much memory is allocated to each level and how many levels are needed. Fortunately, HTMs automatically learn the best possible representations at each level given the statistics of the input and the amount of resources allocated. If you allocate more memory to a level, that level will form representations that are larger and more complex, which in turn means fewer hierarchical levels may be necessary. If you allocate less memory, a level will form representations that are smaller and simpler, which in turn means more hierarchical levels may be needed.

Up to this point we have been describing difficult problems, such as vision inference ("inference" is similar to pattern recognition). But many valuable problems are simpler than vision, and a single HTM region might prove sufficient. For example, we applied an HTM to predicting where a person browsing a website is likely to click next. This problem involved feeding the HTM network streams of web click data. In this problem there was little or no spatial hierarchy, the solution mostly required discovering the temporal statistics, i.e. predicting where the user would click next by recognizing typical user patterns. The temporal learning algorithms in HTMs are ideal for such problems.

In summary, hierarchies reduce training time, reduce memory usage, and introduce a form of generalization. However, many simpler prediction problems can be solved with a single HTM region.

Regions

The notion of regions wired in a hierarchical fashion comes from biology. The neocortex is a large sheet of neural tissue about 2mm thick. Biologists divide the neocortex into a hierarchy of regions primarily based on how the regions connect to each other but also by what the neurons in each region respond to. All neocortical regions look similar in their details. They are comprised of layers of neurons (typically five layers of neurons plus one non-cellular layer). Most connections are between neurons in the same region, and most of those connections are between neurons in the same layer within the region. Cells are arranged in columns that span the layers. All the neurons in a column have similar response properties. This

high-level description of what the neocortex looks like leaves out many details, but it will serve our purposes here.

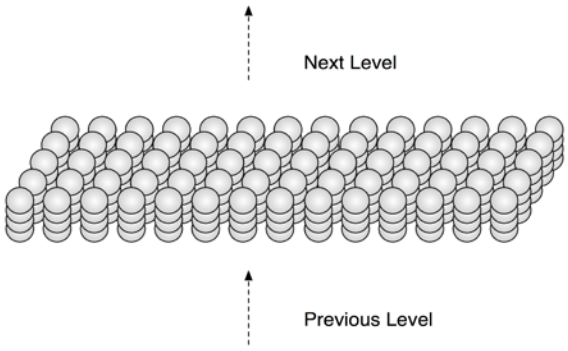

Figure 1.3: A small section of an HTM region, modeling one layer of cells in a region of the neocortex, shown with cells arranged in four-deep columns

HTM regions are comprised of a sheet of highly interconnected cells in a columnar arrangement. A region could be considered similar to one of the five layers of neurons seen in the neocortex. We believe that each layer in the neocortex is implementing a similar learning and prediction function which, through variations in connectivity, play different roles, such as inference, behavior, feedback, and attention.

We propose that the HTM regions described in this document capture many of the mechanisms used in all neocortical layers. We further propose that an HTM region with a single layer of cells is roughly equivalent to the feed-forward layers in the neocortex. Single layer HTM regions can do inference and prediction on complex data streams and therefore should be useful in many problems.

Sparse Distributed Representations

Although neurons in the neocortex are highly interconnected, inhibitory neurons guarantee that only a small percentage of the neurons are active at one time. Thus, information in the brain is always represented by a small percentage of active neurons within a large population of neurons. This kind of encoding is called a "sparse distributed representation". "Sparse" means that only a small percentage of neurons are active at one time. "Distributed" means that the activations of many neurons are required in order to represent something. A single active neuron conveys some meaning but it must be interpreted within the context of a population of neurons to convey the full meaning.

HTM regions also use sparse distributed representations. In fact, the memory mechanisms within an HTM region are dependent on using sparse distributed

representations, and wouldn't work otherwise. The input to an HTM region is always a distributed representation, but it may not be sparse, so the first thing an HTM region does is to convert its input into a sparse distributed representation.

For example, a region might receive 20,000 input bits. The percentage of input bits that are "1" and "0" might vary significantly over time. One time there might be 5,000 "1" bits and another time there might be 9,000 "1" bits. The HTM region could convert this input into an internal representation of 10,000 bits of which 2%, or 200, are active at once, regardless of how many of the input bits are "1". As the input to the HTM region varies over time, the internal representation also will change, but there always will be about 200 bits out of 10,000 active.

It may seem that this process generates a large loss of information as the number of possible input patterns is much greater than the number of possible representations in the region. However, both numbers are incredibly big. The actual inputs seen by a region will be a miniscule fraction of all possible inputs. Later we will describe how a region creates a sparse representation from its input. The theoretical loss of information will not have a practical effect.

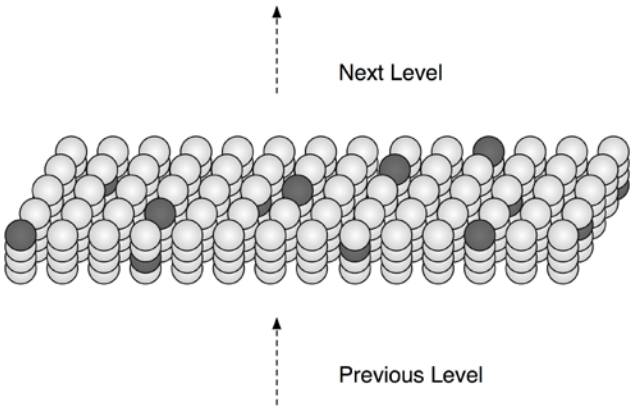

Figure 1.4: An HTM region showing sparse distributed cell activation

Sparse distributed representations have several desirable properties and are integral to the operation of HTMs. They will be touched on again later.

The role of time

Time plays a crucial role in learning, inference, and prediction.

Let's start with inference. Without using time, we can infer almost nothing from our tactile and auditory senses. For example if you are blindfolded and someone places an apple in your hand, you can identify what it is after manipulating it for just a second or so. As you move your fingers over the apple, although the tactile information is constantly changing, the object itself – the apple, as well as your high-

level percept for “apple” – stays constant. However, if an apple was placed on your outstretched palm, and you weren’t allowed to move your hand or fingers, you would have great difficulty identifying it as an apple rather than a lemon.

The same is true for hearing. A static sound conveys little meaning. A word like “apple,” or the crunching sounds of someone biting into an apple, can only be recognized from the dozens or hundreds of rapid, sequential changes over time of the sound spectrum.

Vision, in contrast, is a mixed case. Unlike with touch and hearing, humans are able to recognize images when they are flashed in front of them too fast to give the eyes a chance to move. Thus, visual inference does not always require time-changing inputs. However, during normal vision we constantly move our eyes, heads and bodies, and objects in the world move around us too. Our ability to infer based on quick visual exposure is a special case made possible by the statistical properties of vision and years of training. The general case for vision, hearing, and touch is that inference requires time-changing inputs.

Having covered the general case of inference, and the special case of vision inference of static images, let’s look at learning. In order to learn, all HTM systems must be exposed to time-changing inputs during training. Even in vision, where static inference is sometimes possible, we must see changing images of objects to learn what an object looks like. For example, imagine a dog is running toward you. At each instance in time the dog causes a pattern of activity on the retina in your eye. You perceive these patterns as different views of the same dog, but mathematically the patterns are entirely dissimilar. The brain learns that these different patterns mean the same thing by observing them in sequence. Time is the “supervisor”, teaching you which spatial patterns go together.

Note that it isn’t sufficient for sensory input merely to change. A succession of unrelated sensory patterns would only lead to confusion. The time-changing inputs must come from a common source in the world. Note also that although we use human senses as examples, the general case applies to non-human senses as well. If we want to train an HTM to recognize patterns from a power plant’s temperature, vibration and noise sensors, the HTM will need to be trained on data from those sensors changing through time.

Typically, an HTM network needs to be trained with lots of data. You learned to identify dogs by seeing many instances of many breeds of dogs, not just one single view of one single dog. The job of the HTM algorithms is to learn the temporal sequences from a stream of input data, i.e. to build a model of which patterns follow which other patterns. This job is difficult because it may not know when sequences start and end, there may be overlapping sequences occurring at the same time, learning has to occur continuously, and learning has to occur in the presence of noise.

Learning and recognizing sequences is the basis of forming predictions. Once an HTM learns what patterns are likely to follow other patterns, it can predict the likely next pattern(s) given the current input and immediately past inputs. Prediction is covered in more detail later.

We now will turn to the four basic functions of HTM: learning, inference, prediction, and behavior. Every HTM region performs the first three functions: learning, inference, and prediction. Behavior, however, is different. We know from biology that most neocortical regions have a role in creating behavior but we do not believe it is essential for many interesting applications. Therefore we have not included behavior in our current implementation of HTM. We mention it here for completeness.

Learning

An HTM region learns about its world by finding patterns and then sequences of patterns in sensory data. The region does not "know" what its inputs represent; it works in a purely statistical realm. It looks for combinations of input bits that occur together often, which we call spatial patterns. It then looks for how these spatial patterns appear in sequence over time, which we call temporal patterns or sequences.

If the input to the region represents environmental sensors on a building, the region might discover that certain combinations of temperature and humidity on the north side of the building occur often and that different combinations occur on the south side of the building. Then it might learn that sequences of these combinations occur as each day passes.

If the input to a region represented information related to purchases within a store, the HTM region might discover that certain types of articles are purchased on weekends, or that when the weather is cold certain price ranges are favored in the evening. Then it might learn that different individuals follow similar sequential patterns in their purchases.

A single HTM region has limited learning capability. A region automatically adjusts what it learns based on how much memory it has and the complexity of the input it receives. The spatial patterns learned by a region will necessarily become simpler if the memory allocated to a region is reduced. Or the spatial patterns learned can become more complex if the allocated memory is increased. If the learned spatial patterns in a region are simple, then a hierarchy of regions may be needed to understand complex images. We see this characteristic in the human vision system where the neocortical region receiving input from the retina learns spatial patterns for small parts of the visual space. Only after several levels of hierarchy do spatial patterns combine and represent most or all of the visual space.

Like a biological system, the learning algorithms in an HTM region are capable of "on-line learning", i.e. they continually learn from each new input. There isn't a need for a learning phase separate from an inference phase, though inference improves after additional learning. As the patterns in the input change, the HTM region will gradually change, too.

After initial training, an HTM can continue to learn or, alternatively, learning can be disabled after the training phase. Another option is to turn off learning only at the lowest levels of the hierarchy but continue to learn at the higher levels. Once an HTM has learned the basic statistical structure of its world, most new learning occurs in the upper levels of the hierarchy. If an HTM is exposed to new patterns that have previously unseen low-level structure, it will take longer for the HTM to learn these new patterns. We see this trait in humans. Learning new words in a language you already know is relatively easy. However, if you try to learn new words from a foreign language with unfamiliar sounds, you'll find it much harder because you don't already know the low level sounds.

Simply discovering patterns is a potentially valuable capability. Understanding the high-level patterns in market fluctuations, disease, weather, manufacturing yield, or failures of complex systems, such as power grids, is valuable in itself. Even so, learning spatial and temporal patterns is mostly a precursor to inference and prediction.

Inference

After an HTM has learned the patterns in its world, it can perform inference on novel inputs. When an HTM receives input, it will match it to previously learned spatial and temporal patterns. Successfully matching new inputs to previously stored sequences is the essence of inference and pattern matching.

Think about how you recognize a melody. Hearing the first note in a melody tells you little. The second note narrows down the possibilities significantly but it may still not be enough. Usually it takes three, four, or more notes before you recognize the melody. Inference in an HTM region is similar. It is constantly looking at a stream of inputs and matching them to previously learned sequences. An HTM region can find matches from the beginning of sequences but usually it is more fluid, analogous to how you can recognize a melody starting from anywhere. Because HTM regions use distributed representations, the region's use of sequence memory and inference are more complicated than the melody example implies, but the example gives a flavor for how it works.

It may not be immediately obvious, but every sensory experience you have ever had has been novel, yet you easily find familiar patterns in this novel input. For

example, you can understand the word "breakfast" spoken by almost anyone, no matter whether they are old or young, male or female, are speaking quickly or slowly, or have a strong accent. Even if you had the same person say the same word "breakfast" a hundred times, the sound would never stimulate your cochleae (auditory receptors) in exactly the same way twice.

An HTM region faces the same problem your brain does: inputs may never repeat exactly. Consequently, just like your brain, an HTM region must handle novel input during inference and training. One way an HTM region copes with novel input is through the use of sparse distributed representations. A key property of sparse distributed representations is that you only need to match a portion of the pattern to be confident that the match is significant.

Prediction

Every region of an HTM stores sequences of patterns. By matching stored sequences with current input, a region forms a prediction about what inputs will likely arrive next. HTM regions actually store transitions between sparse distributed representations. In some instances the transitions can look like a linear sequence, such as the notes in a melody, but in the general case many possible future inputs may be predicted at the same time. An HTM region will make different predictions based on context that might stretch back far in time. The majority of memory in an HTM is dedicated to sequence memory, or storing transitions between spatial patterns.

Following are some key properties of HTM prediction.

1) Prediction is continuous.
Without being conscious of it, you are constantly predicting. HTMs do the same. When listening to a song, you are predicting the next note. When walking down the stairs, you are predicting when your foot will touch the next step. When watching a baseball pitcher throw, you are predicting that the ball will come near the batter. In an HTM region, prediction and inference are almost the same thing. Prediction is not a separate step but integral to the way an HTM region works.

2) Prediction occurs in every region at every level of the hierarchy.
If you have a hierarchy of HTM regions, prediction will occur at each level. Regions will make predictions about the patterns they have learned. In a language example, lower level regions might predict possible next phonemes, and higher level regions might predict words or phrases.

3) Predictions are context sensitive.
Predictions are based on what has occurred in the past, as well as what is occurring now. Thus an input will produce different predictions based on previous context.

An HTM region learns to use as much prior context as needed, and can keep the context over both short and long stretches of time. This ability is known as "variable order" memory. For example, think about a memorized speech such as the Gettysburg Address. To predict the next word, knowing just the current word is rarely sufficient; the word "and" is followed by "seven" and later by "dedicated" just in the first sentence. Sometimes, just a little bit of context will help prediction; knowing "four score and" would help predict "seven". Other times, there are repetitive phrases, and one would need to use the context of a far longer timeframe to know where you are in the speech, and therefore what comes next.

4) Prediction leads to stability.

The output of a region is its prediction. One of the properties of HTMs is that the outputs of regions become more stable – that is slower changing, longer-lasting – the higher they are in the hierarchy. This property results from how a region predicts. A region doesn't just predict what will happen immediately next. If it can, it will predict multiple steps ahead in time. Let's say a region can predict five steps ahead. When a new input arrives, the newly predicted step changes but the four of the previously predicted steps might not. Consequently, even though each new input is completely different, only a part of the output is changing, making outputs more stable than inputs. This characteristic mirrors our experience of the real world, where high level concepts – such as the name of a song – change more slowly than low level concepts – the actual notes of the song.

5) A prediction tells us if a new input is expected or unexpected.

Each HTM region is a novelty detector. Because each region predicts what will occur next, it "knows" when something unexpected happens. HTMs can predict many possible next inputs simultaneously, not just one. So it may not be able to predict exactly what will happen next, but if the next input doesn't match any of the predictions the HTM region will know that an anomaly has occurred.

6) Prediction helps make the system more robust to noise.

When an HTM predicts what is likely to happen next, the prediction can bias the system toward inferring what it predicted. For example, if an HTM were processing spoken language, it would predict what sounds, words, and ideas are likely to be uttered next. This prediction helps the system fill in missing data. If an ambiguous sound arrives, the HTM will interpret the sound based on what it is expecting, thus helping inference even in the presence of noise.

In an HTM region, sequence memory, inference, and prediction are intimately integrated. They are the core functions of a region.

Behavior

Our behavior influences what we perceive. As we move our eyes, our retina receives changing sensory input. Moving our limbs and fingers causes varying touch sensation to reach the brain. Almost all our actions change what we sense. Sensory input and motor behavior are intimately entwined.

For decades the prevailing view was that a single region in the neocortex, the primary motor region, was where motor commands originated in the neocortex. Over time it was discovered that most or all regions in the neocortex have a motor output, even low level sensory regions. It appears that all cortical regions integrate sensory and motor functions.

We expect that a motor output could be added to each HTM region within the currently existing framework since generating motor commands is similar to making predictions. However, all the implementations of HTMs to date have been purely sensory, without a motor component.

Progress toward the implementation of HTM

We have made substantial progress turning the HTM theoretical framework into a practical technology. We have implemented and tested several versions of the HTM cortical learning algorithms and have found the basic architecture to be sound. As we test the algorithms on new data sets, we will refine the algorithms and add missing pieces. We will update this document as we do. The next three chapters describe the current state of the algorithms.

There are many components of the theory that are not yet implemented, including attention, feedback between regions, specific timing, and behavior/sensory-motor integration. These missing components should fit into the framework already created.

Chapter 2 – HTM Cortical Learning Algorithms

This chapter describes the learning algorithms at work inside an HTM region. Chapters 3 and 4 describe the implementation of the learning algorithms using pseudocode, whereas this chapter is more conceptual.

Terminology

Before we get started, a note about terminology might be helpful. We use the language of neuroscience in describing the HTM learning algorithms. Terms such as cells, synapses, potential synapses, dendrite segments, and columns are used throughout. This terminology is logical since the learning algorithms were largely derived by matching neuroscience details with theoretical needs. However, in the process of implementing the algorithms we were confronted with performance issues and therefore once we felt we understood how something worked we would look for ways to speed processing. This often involved deviating from a strict adherence to biological details as long as we could get the same results. If you are new to neuroscience this won't be a problem. However, if you are familiar with neuroscience terms, you might find yourself confused as our use of terms varies from your expectation. We intend to write a chapter on the biological mapping of the HTM learning algorithms, but right now it will be helpful to mention a few of the deviations that are likely to cause the most confusion.

Cell states
HTM cells have three output states, active from feed-forward input, active from lateral input, and inactive. The first output state corresponds to a short burst of action potentials. The second output state corresponds to a slower, steady rate of action potentials. We have not found a need for modeling individual action potentials or even scalar rates of activity beyond the two active states. The use of distributed representations seems to overcome the need to model scalar activity rates in cells.

Dendrite segments
HTM cells have a relatively realistic (and therefore complex) dendrite model. In theory each HTM cell has one proximal dendrite segment and a dozen or two distal dendrite segments. The proximal dendrite segment receives feed-forward input and the distal dendrite segments receive lateral input from nearby cells. A class of inhibitory cells forces all the cells in a column to respond to similar feed-forward input. To simplify, we removed the proximal dendrite segment from each cell and replaced it with a single shared dendrite segment per column of cells. The spatial pooler function (described below) operates on the shared dendrite segment, at the level of columns. The temporal pooler function operates on distal dendrite segments, at the level of individual cells within columns. This simplification

achieves the same functionality, though in biology there is no equivalent to a dendrite segment attached to a column.

Synapses
HTM synapses have binary weights. Biological synapses have varying weights but they are also partially stochastic, suggesting a biological neuron cannot rely on precise synaptic weights. The use of distributed representations in HTMs plus our model of dendrite operation allows us to assign binary weights to HTM synapses with no ill effect. To model the forming and un-forming of synapses we use two additional concepts from neuroscience that you may not be familiar with. One is the concept of "potential synapses". This represents all the axons that pass close enough to a dendrite segment that they could potentially form a synapse. The second is called "permanence". This is a scalar value assigned to each potential synapse. The permanence of a synapse represents a range of connectedness between an axon and a dendrite. Biologically, the range would go from completely unconnected, to starting to form a synapse but not connected yet, to a minimally connected synapse, to a large fully connected synapse. The permanence of a synapse is a scalar value ranging from 0 to 1. Learning involves incrementing and decrementing a synapse's permanence. When a synapse's permanence is above a threshold, it is connected with a weight of "1". When it is below the threshold, it is unconnected with a weight of "0".

Overview

Imagine that you are a region of an HTM. Your input consists of thousands or tens of thousands of bits. These input bits may represent sensory data or they may come from another region lower in the hierarchy. They are turning on and off in complex ways. What are you supposed to do with this input?

We already have discussed the answer in its simplest form. Each HTM region looks for common patterns in its input and then learns sequences of those patterns. From its memory of sequences, each region makes predictions. That high level description makes it sound easy, but in reality there is a lot going on. Let's break it down a little further into the following three steps:

1) Form a sparse distributed representation of the input
2) Form a representation of the input in the context of previous inputs
3) Form a prediction based on the current input in the context of previous inputs

We will discuss each of these steps in more detail.

1) Form a sparse distributed representation of the input

When you imagine an input to a region, think of it as a large number of bits. In a brain these would be axons from neurons. At any point in time some of these input bits will be active (value 1) and others will be inactive (value 0). The percentage of input bits that are active vary, say from 0% to 60%. The first thing an HTM region does is to convert this input into a new representation that is sparse. For example, the input might have 40% of its bits "on" but the new representation has just 2% of its bits "on".

An HTM region is logically comprised of a set of *columns*. Each column is comprised of one or more *cells*. Columns may be logically arranged in a 2D array but this is not a requirement. Each column in a region is connected to a unique subset of the input bits (usually overlapping with other columns but never exactly the same subset of input bits). As a result, different input patterns result in different levels of activation of the columns. The columns with the strongest activation inhibit, or deactivate, the columns with weaker activation. (The inhibition occurs within a radius that can span from very local to the entire region.) The sparse representation of the input is encoded by which columns are active and which are inactive after inhibition. The inhibition function is defined to achieve a relatively constant percentage of columns to be active, even when the number of input bits that are active varies significantly.

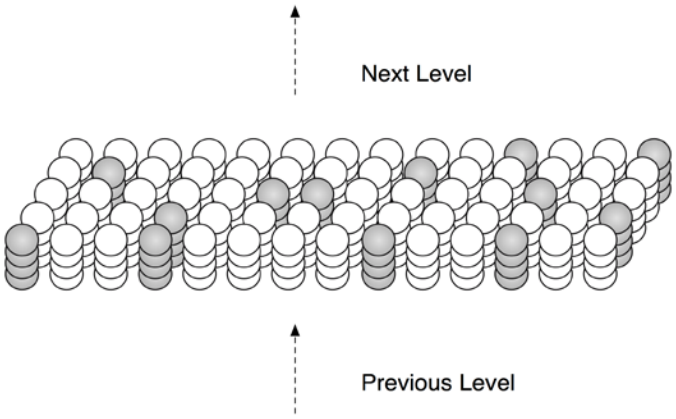

Figure 2.1: An HTM region consists of columns of cells. Only a small portion of a region is shown. Each column of cells receives activation from a unique subset of the input. Columns with the strongest activation inhibit columns with weaker activation. The result is a sparse distributed representation of the input (active columns are shown in light grey).

Imagine now that the input pattern changes. If only a few input bits change, some columns will receive a few more or a few less inputs in the "on" state, but the set of active columns will not likely change much. Thus similar input patterns (ones that have a significant number of active bits in common) will map to a relatively stable set of active columns. How stable the encoding is depends greatly on what inputs each column is connected to. These connections are learned via a method described later.

All these steps (learning the connections to each column from a subset of the inputs, determining the level of input to each column, and using inhibition to select a sparse set of active columns) is referred to as the "Spatial Pooler". The term means patterns that are "spatially" similar (meaning they share a large number of active bits) are "pooled" (meaning they are grouped together in a common representation).

2) Form a representation of the input in the context of previous inputs

The next function performed by a region is to convert the columnar representation of the input into a new representation that includes state, or context, from the past. The new representation is formed by activating a subset of the cells within each column, typically only one cell per column.

Consider hearing two spoken sentences, "I ate a pear" and "I have eight pears". The words "ate" and "eight" are homonyms; they sound identical. We can be certain that at some point in the brain there are neurons that respond identically to the spoken words "ate" and "eight". After all, identical sounds are entering the ear. However, we also can be certain that at another point in the brain the neurons that respond to this input are different, in different contexts. The representations for the sound "ate" will be different when you hear "I ate" vs. "I have eight". Imagine that you have memorized the two sentences "I ate a pear" and "I have eight pears". Hearing "I ate…" leads to a different prediction than "I have eight…". There must be different internal representations after hearing "I ate" and "I have eight".

This principle of encoding an input differently in different contexts is a universal feature of perception and action and is one of the most important functions of an HTM region. It is hard to overemphasize the importance of this capability.

Each column in an HTM region consists of multiple cells. Each cell in a column can be active or not active. By selecting different active cells in each active column, we can represent the exact same input differently in different contexts. A specific example might help. Say every column has 4 cells and the representation of every input consists of 100 active columns. If only one cell per column is active at a time, we have 4^100 ways of representing the exact same input. The same input will always result in the same 100 columns being active, but in different contexts different cells in those columns will be active. Now we can represent the same input in a very large number of contexts, but how unique will those different representations be? Nearly all randomly chosen pairs of the 4^100 possible patterns will overlap by about 25 cells. Thus two representations of a particular input in different contexts will have about 25 cells in common and 75 cells that are different, making them easily distinguishable.

The general rule used by an HTM region is the following. When a column becomes active, it will activate all the cells in the column if the input was unexpected. If one

or more cells in the column are in the predictive state, only those cells will become active, the rest of the cells in the column remain inactive.

If there is no prior state, and therefore no context and prediction, all the cells in a column will become active when the column becomes active. This scenario is similar to hearing the first note in a song. Without context you usually can't predict what will happen next; all options are available. If there is prior state but the input does not match what is expected, all the cells in the active column will become active. This determination is done on a column by column basis so a predictive match or mismatch is never an "all-or-nothing" event.

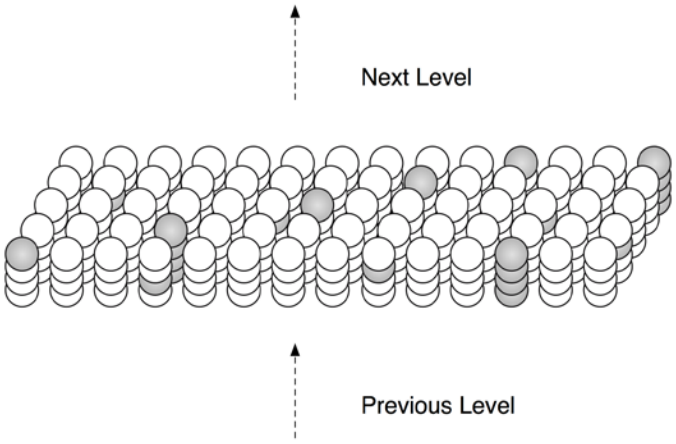

Figure 2.2: By activating a subset of cells in each column, an HTM region can represent the same input in many different contexts. Columns only activate predicted cells. Columns with no predicted cells activate all the cells in the column. The figure shows some columns with one cell active and some columns with all cells active.

.

As mentioned in the terminology section above, HTM cells can be in one of three states. If a cell is active due to feed-forward input we just use the term "active". If the cell is active due to lateral connections to other nearby cells we say it is in the "predictive state".

3) Form a prediction based on the input in the context of previous inputs

The final step for our region is to make a prediction of what is likely to happen next. The prediction is based on the representation formed in step 2), which includes context from all previous inputs.

When a region makes a prediction it activates (into the predictive state) all the cells that will likely become active due to future feed-forward input. Because representations in a region are sparse, multiple predictions can be made at the same time. For example if 2% of the columns are active due to an input, you could expect that ten different predictions could be made resulting in 20% of the columns having a predicted cell. Or, twenty different predictions could be made resulting in 40% of

the columns having a predicted cell. If each column had four cells, with one active at a time, then 10% of the cells would be in the predictive state.

A future chapter on sparse distributed representations will show that even though different predictions are merged together, a region can know with high certainty whether a particular input was predicted or not.

How does a region make a prediction? When input patterns change over time, different sets of columns and cells become active in sequence. When a cell becomes active, it forms connections to a subset of the cells nearby that were active immediately prior. These connections can be formed quickly or slowly depending on the learning rate required by the application. Later, all a cell needs to do is to look at these connections for coincident activity. If the connections become active, the cell can expect that it might become active shortly and enters a predictive state. Thus the feed-forward activation of a set of cells will lead to the predictive activation of other sets of cells that typically follow. Think of this as the moment when you recognize a song and start predicting the next notes.

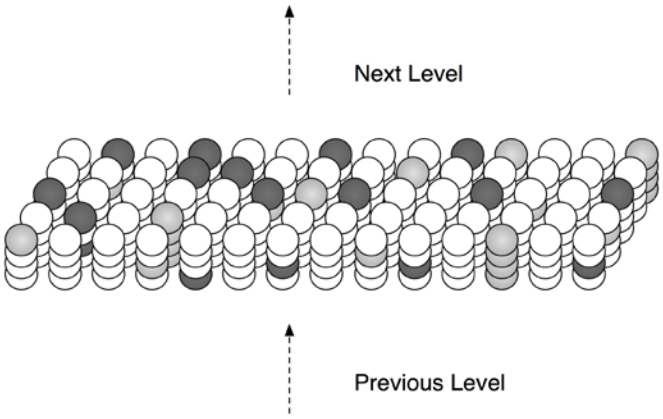

Figure 2.3: At any point in time, some cells in an HTM region will be active due to feed-forward input (shown in light gray). Other cells that receive lateral input from active cells will be in a predictive state (shown in dark gray).

In summary, when a new input arrives, it leads to a sparse set of active columns. One or more of the cells in each column become active, these in turn cause other cells to enter a predictive state through learned connections between cells in the region. The cells activated by connections within the region constitute a prediction of what is likely to happen next. When the next feed-forward input arrives, it selects another sparse set of active columns. If a newly active column is unexpected, meaning it was not predicted by any cells, it will activate all the cells in the columns. If a newly active column has one or more predicted cells, only those cells will become active. The output of a region is the activity of all cells in the region, including the cells active because of feed-forward input *and* the cells active in the predictive state.

As mentioned earlier, predictions are not just for the *next* time step. Predictions in an HTM region can be for several time steps into the future. Using melodies as example, an HTM region would not just predict the next note in a melody, but might predict the next four notes. This leads to a desirable property. The output of a region (the union of all the active and predicted cells in a region) changes more slowly than the input. Imagine the region is predicting the next four notes in a melody. We will represent the melody by the letter sequence A,B,C,D,E,F,G. After hearing the first two notes, the region recognizes the sequence and starts predicting. It predicts C,D,E,F. The "B" cells are already active so cells for B,C,D,E,F are all in one of the two active states. Now the region hears the next note "C". The set of active and predictive cells now represents "C,D,E,F,G". Note that the input pattern changed completely going from "B" to "C", but only 20% of the cells changed.

Because the output of an HTM region is a vector representing the activity of all the region's cells, the output in this example is five times more stable than the input. In a hierarchical arrangement of regions, we will see an increase in temporal stability as you ascend the hierarchy.

We use the term "temporal pooler" to describe the two steps of adding context to the representation and predicting. By creating slowly changing outputs for sequences of patterns, we are in essence "pooling" together different patterns that follow each other in time.

Now we will go into another level of detail. We start with concepts that are shared by the spatial pooler and temporal pooler. Then we discuss concepts and details unique to the spatial pooler followed by concepts and details unique to the temporal pooler.

Shared concepts

Learning in the spatial pooler and temporal pooler are similar. Learning in both cases involves establishing connections, or synapses, between cells. The temporal pooler learns connections between cells in the same region. The spatial pooler learns feed-forward connections between input bits and columns.

Binary weights

HTM synapses have only a 0 or 1 effect; their "weight" is binary, a property unlike many neural network models which use scalar variable values in the range of 0 to 1.

Permanence

Synapses are forming and unforming constantly during learning. We assign a scalar value to each synapse (0.0 to 1.0) to indicate how permanent the connection is. When a connection is reinforced, its permanence is increased. Under other

conditions, the permanence is decreased. When the permanence is above a threshold (e.g. 0.2), the synapse is considered to be established. If the permanence is below the threshold, the synapse will have no effect.

Dendrite segments
Synapses connect to dendrite segments. There are two types of dendrite segments.
- One type of dendrite segment forms synapses with feed-forward inputs. The active synapses on this type of segment are linearly summed to determine the feed-forward activation of a column.
- The other type of dendrite segment forms synapses with cells within the region. Every cell has several dendrite segments of this type. If the sum of the active synapses on this type of segment exceeds a threshold, then the associated cell becomes active in a predicted state. Since there are multiple dendrite segments per cell, a cell's predictive state is the logical OR operation of several constituent threshold detectors.

Potential Synapses
The concept of potential synapses is also borrowed from neuroscience. It refers to the fact that not all connections are physically possible. Therefore each dendrite segment has a list of potential synapses. All the potential synapses are given a permanence value and may become functional synapses if their permanence values exceed a threshold.

Learning
Learning involves incrementing or decrementing the permanence values of potential synapses on a dendrite segment. The rules used for making synapses more or less permanent are similar to "Hebbian" learning rules. For example, if a post-synaptic cell is active due to a dendrite segment receiving input above its threshold, then the permanence values of the synapses on that segment are modified. Synapses that are active, and therefore contributed to the cell being active, have their permanence increased. Synapses that are inactive, and therefore did not contribute, have their permanence decreased. The exact conditions under which synapse permanence values are updated differ in the spatial and temporal pooler. The details are described below.

Now we will discuss concepts specific to the spatial and temporal pooler functions.

Spatial pooler concepts

The most fundamental function of the spatial pooler is to convert a region's input into a sparse pattern. This function is important because the mechanism used to learn sequences and make predictions requires starting with sparse distributed patterns.

There are several overlapping goals for the spatial pooler, which determine how the spatial pooler operates and learns.

1) Use all columns
An HTM region has a fixed number of columns that learn to represent common patterns in the input. One objective is to make sure all the columns learn to represent something useful regardless of how many columns you have. We don't want columns that are never active. To prevent this from happening, we keep track of how often a column is active relative to its neighbors. If the relative activity of a column is too low, it boosts its input activity level until it starts to be part of the winning set of columns. In essence, all columns are competing with their neighbors to be a participant in representing input patterns. If a column is not very active, it will become more aggressive. When it does, other columns will be forced to modify their input and start representing slightly different input patterns.

2) Maintain desired density
A region needs to form a sparse representation of its inputs. Columns with the most input inhibit their neighbors. There is a radius of inhibition which is proportional to the size of the receptive fields of the columns (and therefore can range from small to the size of the entire region). Within the radius of inhibition, we allow only a percentage of the columns with the most active input to be "winners". The remainders of the columns are disabled. (A "radius" of inhibition implies a 2D arrangement of columns, but the concept can be adapted to other topologies.)

3) Avoid trivial patterns
We want all our columns to represent non-trivial patterns in the input. This goal can be achieved by setting a minimum threshold of input for the column to be active. For example, if we set the threshold to 50, it means that a column must have a least 50 active synapses on its dendrite segment to be active, guaranteeing a certain level of complexity to the pattern it represents.

4) Avoid extra connections
If we aren't careful, a column could form a large number of valid synapses. It would then respond strongly to many different unrelated input patterns. Different subsets of the synapses would respond to different patterns. To avoid this problem, we decrement the permanence value of any synapse that isn't currently contributing to a winning column. By making sure non-contributing synapses are sufficiently penalized, we guarantee a column represents a limited number input patterns, sometimes only one.

5) Self adjusting receptive fields
Real brains are highly "plastic"; regions of the neocortex can learn to represent entirely different things in reaction to various changes. If part of the neocortex is damaged, other parts will adjust to represent what the damaged part used to represent. If a sensory organ is damaged or changed, the associated part of the neocortex will adjust to represent something else. The system is self-adjusting.

We want our HTM regions to exhibit the same flexibility. If we allocate 10,000 columns to a region, it should learn how to best represent the input with 10,000 columns. If we allocate 20,000 columns, it should learn how best to use that number. If the input statistics change, the columns should change to best represent the new reality. In short, the designer of an HTM should be able to allocate any resources to a region and the region will do the best job it can of representing the input based on the available columns and input statistics. The general rule is that with more columns in a region, each column will represent larger and more detailed patterns in the input. Typically the columns also will be active less often, yet we will maintain a relative constant sparsity level.

No new learning rules are required to achieve this highly desirable goal. By boosting inactive columns, inhibiting neighboring columns to maintain constant sparsity, establishing minimal thresholds for input, maintaining a large pool of potential synapses, and adding and forgetting synapses based on their contribution, the ensemble of columns will dynamically configure to achieve the desired effect.

Spatial pooler details

We can now go through everything the spatial pooling function does.

1) Start with an input consisting of a fixed number of input bits. These bits might represent sensory data or they might come from another region lower in the hierarchy.

2) Assign a fixed number of columns to the region receiving this input. Each column has an associated dendrite segment. Each dendrite segment has a set of potential synapses representing a subset of the input bits. Each potential synapse has a permanence value. Based on their permanence values, some of the potential synapses will be valid.

3) For any given input, determine how many valid synapses on each column are connected to active input bits.

4) The number of active synapses is multiplied by a "boosting" factor which is dynamically determined by how often a column is active relative to its neighbors.

5) The columns with the highest boosted input disable all but a fixed percentage of the columns within an inhibition radius. The inhibition radius is itself dynamically determined by the spread (or "fan-out") of input bits. There is now a sparse set of active columns.

6) For each of the active columns, we adjust the permanence values of all the potential synapses. The permanence values of synapses aligned with active input bits are increased. The permanence values of synapses aligned with inactive input bits are decreased. The changes made to permanence values may change some synapses from being valid to not valid, and vice-versa.

Temporal pooler concepts

Recall that the temporal pooler learns sequences and makes predictions. The basic method is that when a cell becomes active, it forms connections to other cells that were active just prior. Cells can then predict when they will become active by looking at their connections. If all the cells do this, collectively they can store and recall sequences, and they can predict what is likely to happen next. There is no central storage for a sequence of patterns; instead, memory is distributed among the individual cells. Because the memory is distributed, the system is robust to noise and error. Individual cells can fail, usually with little or no discernible effect.

It is worth noting a few important properties of sparse distributed representations that the temporal pooler exploits.

Assume we have a hypothetical region that always forms representations by using 200 active cells out of a total of 10,000 cells (2% of the cells are active at any time). How can we remember and recognize a particular pattern of 200 active cells? A simple way to do this is to make a list of the 200 active cells we care about. If we see the same 200 cells active again we recognize the pattern. However, what if we made a list of only 20 of the 200 active cells and ignored the other 180? What would happen? You might think that remembering only 20 cells would cause lots of errors, that those 20 cells would be active in many different patterns of 200. But this isn't the case. Because the patterns are large and sparse (in this example 200 active cells out of 10,000), remembering 20 active cells is almost as good as remembering all 200. The chance for error in a practical system is exceedingly small.

The cells in an HTM region take advantage of this property. Each of a cell's dendrite segments has a set of connections to other cells in the region. A dendrite segment forms these connections as a means of recognizing the state of the network at some point in time. There may be hundreds or thousands of active cells nearby but the dendrite segment only has to connect to 15 or 20 of them. When the dendrite segment sees 15 of those active cells, it can be fairly certain the larger pattern is occurring. This technique is called "sub-sampling" and is used throughout the HTM algorithms.

Every cell participates in many different distributed patterns and in many different sequences. A particular cell might be part of dozens or hundreds of temporal transitions. Therefore every cell has several dendrite segments, not just one.

Ideally a cell would have one dendrite segment for each pattern of activity it wants to recognize. Practically though, a dendrite segment can learn connections for several completely different patterns and still work well. For example, one segment might learn 20 connections for each of 4 different patterns, for a total of 80 connections. We set a threshold so the dendrite segment becomes active when any 15 of its connections are active. This introduces the possibility for error as connections from different patterns might incorrectly combine to exceed the threshold. However, this kind of error is very unlikely, again due to the sparseness of the representations.

Now we can see how a cell with one or two dozen dendrite segments and a few thousand synapses can recognize hundreds of separate states of cell activity.

Temporal pooler details

Here we enumerate the steps performed by the temporal pooler. We start where the spatial pooler left off, with a set of active columns representing the feed-forward input.

1) For each active column, check for cells in the column that are in a predictive state, and activate them. If no cells are in a predictive state, activate all the cells in the column. The resulting set of active cells is the representation of the input in the context of prior input.

2) For every dendrite segment on every cell in the region, count how many established synapses are connected to active cells. If the number exceeds a threshold, that dendrite segment is marked as active. Cells with active dendrite segments are put in the predictive state unless they are already active due to feed-forward input. Cells with no active dendrites and not active due to bottom-up input become or remain inactive. The collection of cells now in the predictive state is the prediction of the region.

3) When a dendrite segment becomes active, modify the permanence values of all the synapses associated with the segment. For every potential synapse on the active dendrite segment, increase the permanence of those synapses that are connected to active cells and decrement the permanence of those synapses connected to inactive cells. These changes to synapse permanence are marked as temporary.

This modifies the synapses on segments that are already trained sufficiently to make the segment active, and thus lead to a prediction. However, we always want to extend predictions further back in time if possible. Thus, we pick a second dendrite segment on the same cell to train. For the second segment we choose the one that best matches the state of the system in the previous time step. For this segment, using the state of the system in the previous time step, increase the permanence of

those synapses that are connected to active cells and decrement the permanence of those synapses connected to inactive cells. These changes to synapse permanence are marked as temporary.

4) Whenever a cell switches from being inactive to active due to feed-forward input, we traverse each potential synapse associated with the cell and remove any temporary marks. Thus we update the permanence of synapses only if they correctly predicted the feed-forward activation of the cell.

5) When a cell switches from either active state to inactive, undo any permanence changes marked as temporary for each potential synapse on this cell. We don't want to strengthen the permanence of synapses that incorrectly predicted the feed-forward activation of a cell.

Note that only cells that are active due to feed-forward input propagate activity *within* the region, otherwise predictions would lead to further predictions. But all the active cells (feed-forward and predictive) form the output of a region and propagate to the *next* region in the hierarchy.

First order versus variable order sequences and prediction

There is one more major topic to discuss before we end our discussion on the spatial and temporal poolers. It may not be of interest to all readers and it is not needed to understand Chapters 3 and 4.

What is the effect of having more or fewer cells per column? Specifically, what happens if we have only one cell per column? In the example used earlier, we showed that a representation comprised of 100 active columns with 4 cells per column can be encoded in 4^100 different ways, creating a very big number. Therefore the same input can be used in a great many different contexts. If each input pattern represented a word, then a region could remember many sentences that use the same words over and over again and not get confused. Alternately we could use the same number of words to learn fewer but longer sentences. This kind of memory is called "variable order", meaning that the amount of prior context needed to predict the next input varies. An HTM region is a variable order memory.

If we increase to five cells per column, the available number of encodings of any particular input in our example would increase to 5^100, a huge increase over 4^100. But both these numbers are so large that for many practical problems the increase in capacity might not be useful.

However, making the number of cells per column much smaller does make a big difference.

If we go all the way to one cell per column, we lose the ability to include context in our representations. An input to a region always results in the same prediction, regardless of previous activity. With one cell per column, the memory of an HTM region is a "first order" memory; predictions are based only on the current input.

First order prediction is ideally suited for one type of problem that brains solve: static spatial inference. As stated earlier, a human exposed to a brief visual image can recognize what the object is even if the exposure is too short for the eyes to move. With hearing, you always need to hear a sequence of patterns to recognize what it is. Vision is usually like that, you usually process a stream of visual images. But under certain conditions you can recognize an image with a single exposure.

Temporal and static recognition might appear to require different inference mechanisms. One requires recognizing sequences of patterns and making predictions based on variable length context. The other requires recognizing a static spatial pattern without using temporal context. An HTM region with multiple cells per column is ideally suited for recognizing time-based sequences, and an HTM region with one cell per column is ideally suited to recognizing spatial patterns. At Numenta, we have performed many experiments using one-cell-per-column regions applied to vision problems. The details of these experiments are beyond the scope of this chapter; however we will cover the important concepts.

If we expose an HTM region to images, the columns in the region learn to represent common spatial arrangements of pixels. The kind of patterns learned are similar to what is observed in region V1 in neocortex (a neocortical region extensively studied in biology), typically lines and corners at different orientations. By training on moving images, the HTM region learns transitions of these basic shapes. For example, a vertical line at one position is often followed by a vertical line shifted to the left or right. All the commonly observed transitions of patterns are remembered by the HTM region.

Now what happens if we expose a region to an image of a vertical line moving to the right? If our region has only one cell per column, it will predict the line might next appear to the left or to the right. It can't use the context of knowing where the line was in the past and therefore know if it is moving left or right. What you find is that these one-cell-per-column cells behave like "simple cells" in the neocortex. The predictive output of such a cell will be active for a visible line in different positions, regardless of whether the line is moving left or right or not at all. We have further observed that a region like this exhibits stability to translation, changes in scale, etc. while maintaining the ability to distinguish between different images. This behavior is what is needed for spatial invariance (recognizing the same pattern in different locations of an image).

If we now do the same experiment on an HTM region with multiple cells per column, we find that the cells behave like "complex cells" in the neocortex. The predictive

output of a cell will be active for a line moving to the left or a line moving to the right, but not both.

Putting this all together, we make the following hypothesis. The neocortex has to do both first order and variable order inference and prediction. There are four or five layers of cells in each region of the neocortex. The layers differ in several ways but they all have shared columnar response properties and large horizontal connectivity within the layer. We speculate that each layer of cells in neocortex is performing a variation of the HTM inference and learning rules described in this chapter. The different layers of cells play different roles. For example it is known from anatomical studies that layer 6 creates feedback in the hierarchy and layer 5 is involved in motor behavior. The two primary feed-forward layers of cells are layers 4 and 3. We speculate that one of the differences between layers 4 and 3 is that the cells in layer 4 are acting independently, i.e. one cell per column, whereas the cells in layer 3 are acting as multiple cells per column. Thus regions in the neocortex near sensory input have both first order and variable order memory. The first order sequence memory (roughly corresponding to layer 4 neurons) is useful in forming representations that are invariant to spatial changes. The variable order sequence memory (roughly corresponding to layer 3 neurons) is useful for inference and prediction of moving images.

In summary, we hypothesize that the algorithms similar to those described in this chapter are at work in all layers of neurons in the neocortex. The layers in the neocortex vary in significant details which make them play different roles related to feed-forward vs. feedback, attention, and motor behavior. In regions close to sensory input, it is useful to have a layer of neurons performing first order memory as this leads to spatial invariance.

At Numenta, we have experimented with first order (single cell per column) HTM regions for image recognition problems. We also have experimented with variable order (multiple cells per column) HTM regions for recognizing and predicting variable order sequences. In the future, it would be logical to try to combine these in a single region and to extend the algorithms to other purposes. However, we believe many interesting problems can be addressed with the equivalent of single-layer, multiple-cell-per-column regions, either alone or in a hierarchy.

Chapter 3 – Spatial Pooling Implementation and Pseudocode

This chapter contains the detailed pseudocode for a first implementation of the spatial pooler function. The input to this code is an array of bottom-up binary inputs from sensory data or the previous level. The code computes activeColumns(t) - the list of columns that win due to the bottom-up input at time t. This list is then sent as input to the temporal pooler routine described in the next chapter, i.e. activeColumns(t) is the output of the spatial pooling routine.

The pseudocode is split into three distinct phases that occur in sequence:

Phase 1: compute the overlap with the current input for each column
Phase 2: compute the winning columns after inhibition
Phase 3: update synapse permanence and internal variables

Although spatial pooler learning is inherently online, you can turn off learning by simply skipping Phase 3.

The rest of the chapter contains the pseudocode for each of the three steps. The various data structures and supporting routines used in the code are defined at the end.

Initialization

Prior to receiving any inputs, the region is initialized by computing a list of initial potential synapses for each column. This consists of a random set of inputs selected from the input space. Each input is represented by a synapse and assigned a random permanence value. The random permanence values are chosen with two criteria. First, the values are chosen to be in a small range around connectedPerm (the minimum permanence value at which a synapse is considered "connected"). This enables potential synapses to become connected (or disconnected) after a small number of training iterations. Second, each column has a natural center over the input region, and the permanence values have a bias towards this center (they have higher values near the center).

Phase 1: Overlap

Given an input vector, the first phase calculates the overlap of each column with that vector. The overlap for each column is simply the number of connected synapses with active inputs, multiplied by its boost. If this value is below minOverlap, we set the overlap score to zero.

```
1.  for c in columns
2.
3.          overlap(c) = 0
4.          for s in connectedSynapses(c)
5.                  overlap(c) = overlap(c) + input(t, s.sourceInput)
6.
7.          if overlap(c) < minOverlap then
8.                  overlap(c) = 0
9.          else
10.         overlap(c) = overlap(c) * boost(c)
```

Phase 2: Inhibition

The second phase calculates which columns remain as winners after the inhibition step. desiredLocalActivity is a parameter that controls the number of columns that end up winning. For example, if desiredLocalActivity is 10, a column will be a winner if its overlap score is greater than the score of the 10'th highest column within its inhibition radius.

```
11. for c in columns
12.
13.     minLocalActivity = kthScore(neighbors(c), desiredLocalActivity)
14.
15.     if overlap(c) > 0 and overlap(c) ≥ minLocalActivity then
16.         activeColumns(t).append(c)
17.
```

Phase 3: Learning

The third phase performs learning; it updates the permanence values of all synapses as necessary, as well as the boost and inhibition radius.

The main learning rule is implemented in lines 20-26. For winning columns, if a synapse is active, its permanence value is incremented, otherwise it is decremented. Permanence values are constrained to be between 0 and 1.

Lines 28-36 implement boosting. There are two separate boosting mechanisms in place to help a column learn connections. If a column does not win often enough (as measured by activeDutyCycle), its overall boost value is increased (line 30-32). Alternatively, if a column's connected synapses do not overlap well with any inputs often enough (as measured by overlapDutyCycle), its permanence values are boosted (line 34-36). Note: once learning is turned off, boost(c) is frozen.

Finally, at the end of Phase 3 the inhibition radius is recomputed (line 38).

```
18. for c in activeColumns(t)
19.
20.       for s in potentialSynapses(c)
21.             if active(s) then
22.                   s.permanence += permanenceInc
23.                   s.permanence = min(1.0, s.permanence)
24.             else
25.                   s.permanence -= permanenceDec
26.                   s.permanence = max(0.0, s.permanence)
27.
28. for c in columns:
29.
30.       minDutyCycle(c) = 0.01 * maxDutyCycle(neighbors(c))
31.       activeDutyCycle(c) = updateActiveDutyCycle(c)
32.       boost(c) = boostFunction(activeDutyCycle(c), minDutyCycle(c))
33.
34.       overlapDutyCycle(c) = updateOverlapDutyCycle(c)
35.       if overlapDutyCycle(c) < minDutyCycle(c) then
36.             increasePermanences(c, 0.1*connectedPerm)
37.
38. inhibitionRadius = averageReceptiveFieldSize()
39.
```

Supporting data structures and routines

The following variables and data structures are used in the pseudocode:

columns	List of all columns.
input(t,j)	The input to this level at time t. input(t, j) is 1 if the j'th input is on.
overlap(*c*)	The spatial pooler overlap of column c with a particular input pattern.
activeColumns(*t*)	List of column indices that are winners due to bottom-up input.
desiredLocalActivity	A parameter controlling the number of columns that will be winners after the inhibition step.
inhibitionRadius	Average connected receptive field size of the columns.
neighbors(c)	A list of all the columns that are within inhibitionRadius of column c.
minOverlap	A minimum number of inputs that must be active for a column to be considered during the inhibition step.
boost(c)	The boost value for column c as computed during learning - used to increase the overlap value for inactive columns.
synapse	A data structure representing a synapse - contains a permanence value and the source input index.
connectedPerm	If the permanence value for a synapse is greater than this value, it is said to be connected.
potentialSynapses(c)	The list of potential synapses and their permanence values.
connectedSynapses(c)	A subset of potentialSynapses(c) where the permanence value is greater than connectedPerm. These are the bottom-up inputs that are currently connected to column c.
permanenceInc	Amount permanence values of synapses are incremented during learning.
permanenceDec	Amount permanence values of synapses are decremented during learning.

activeDutyCycle(c)	A sliding average representing how often column c has been active after inhibition (e.g. over the last 1000 iterations).
overlapDutyCycle(c)	A sliding average representing how often column c has had significant overlap (i.e. greater than minOverlap) with its inputs (e.g. over the last 1000 iterations).
minDutyCycle(c)	A variable representing the minimum desired firing rate for a cell. If a cell's firing rate falls below this value, it will be boosted. This value is calculated as 1% of the maximum firing rate of its neighbors.

The following supporting routines are used in the above code.

kthScore(cols, k)

Given the list of columns, return the k'th highest overlap value.

updateActiveDutyCycle(c)

Computes a moving average of how often column c has been active after inhibition.

updateOverlapDutyCycle(c)

Computes a moving average of how often column c has overlap greater than minOverlap.

averageReceptiveFieldSize()

The radius of the average connected receptive field size of all the columns. The connected receptive field size of a column includes only the connected synapses (those with permanence values >= connectedPerm). This is used to determine the extent of lateral inhibition between columns.

maxDutyCycle(cols)

Returns the maximum active duty cycle of the columns in the given list of columns.

increasePermanences(c, s)

Increase the permanence value of every synapse in column c by a scale factor s.

boostFunction(c)

Returns the boost value of a column. The boost value is a scalar >= 1. If activeDutyCyle(c) is above minDutyCycle(c), the boost value is 1. The boost increases linearly once the column's activeDutyCyle starts falling below its minDutyCycle.

Chapter 4 – Temporal Pooling Implementation and Pseudocode

This chapter contains the detailed pseudocode for a first implementation of the temporal pooler function. The input to this code is activeColumns(t), as computed by the spatial pooler. The code computes the active and predictive state for each cell at the current timestep, t. The boolean OR of the active and predictive states for each cell forms the output of the temporal pooler for the next level.

The pseudocode is split into three distinct phases that occur in sequence:

Phase 1: compute the active state, activeState(t), for each cell
Phase 2: compute the predicted state, predictiveState(t), for each cell
Phase 3: update synapses

Phase 3 is only required for learning. However, unlike spatial pooling, Phases 1 and 2 contain some learning-specific operations when learning is turned on. Since temporal pooling is significantly more complicated than spatial pooling, we first list the inference-only version of the temporal pooler, followed by a version that combines inference and learning. A description of some of the implementation details, terminology, and supporting routines are at the end of the chapter, after the pseudocode.

Temporal pooler pseudocode: inference alone

Phase 1

The first phase calculates the active state for each cell. For each winning column we determine which cells should become active. If the bottom-up input was predicted by any cell (i.e. its predictiveState was 1 due to a sequence segment in the previous time step), then those cells become active (lines 4-9). If the bottom-up input was unexpected (i.e. no cells had predictiveState output on), then each cell in the column becomes active (lines 11-13).

```
1.  for c in activeColumns(t)
2.
3.      buPredicted = false
4.      for i = 0 to cellsPerColumn - 1
5.          if predictiveState(c, i, t-1) == true then
6.              s = getActiveSegment(c, i, t-1, activeState)
7.              if s.sequenceSegment == true then
8.                  buPredicted = true
9.                  activeState(c, i, t) = 1
10.
11.     if buPredicted == false then
12.         for i = 0 to cellsPerColumn - 1
13.             activeState(c, i, t) = 1
```

Phase 2

The second phase calculates the predictive state for each cell. A cell will turn on its predictiveState if any one of its segments becomes active, i.e. if enough of its horizontal connections are currently firing due to feed-forward input.

```
14. for c, i in cells
15.     for s in segments(c, i)
16.         if segmentActive(c, i, s, t) then
17.             predictiveState(c, i, t) = 1
```

Temporal pooler pseudocode: combined inference and learning

Phase 1

The first phase calculates the activeState for each cell that is in a winning column. For those columns, the code further selects one cell per column as the learning cell (learnState). The logic is as follows: if the bottom-up input was predicted by any cell (i.e. its predictiveState output was 1 due to a sequence segment), then those cells become active (lines 5-10). If that segment became active from cells chosen with learnState on, this cell is selected as the learning cell (lines 11-13). If the bottom-up input was not predicted, then all cells in the become active (lines 15-17). In addition, the best matching cell is chosen as the learning cell (lines 19-24) and a new segment is added to that cell.

```
18. for c in activeColumns(t)
19.
20.       buPredicted = false
21.       lcChosen = false
22.       for i = 0 to cellsPerColumn - 1
23.             if predictiveState(c, i, t-1) == true then
24.                   s = getActiveSegment(c, i, t-1, activeState)
25.                   if s.sequenceSegment == true then
26.                         buPredicted = true
27.                         activeState(c, i, t) = 1
28.                         if segmentActive(s, t-1, learnState) then
29.                               lcChosen = true
30.                               learnState(c, i, t) = 1
31.
32.       if buPredicted == false then
33.             for i = 0 to cellsPerColumn - 1
34.                   activeState(c, i, t) = 1
35.
36.       if lcChosen == false then
37.             i = getBestMatchingCell(c, t-1)
38.             learnState(c, i, t) = 1
39.             sUpdate = getSegmentActiveSynapses (c, i, -1, t-1, true)
40.             sUpdate.sequenceSegment = true
41.             segmentUpdateList.add(sUpdate)
```

Phase 2

The second phase calculates the predictive state for each cell. A cell will turn on its predictive state output if one of its segments becomes active, i.e. if enough of its lateral inputs are currently active due to feed-forward input. In this case, the cell queues up the following changes: a) reinforcement of the currently active segment (lines 30-31), and b) reinforcement of a segment that could have predicted this activation, i.e. a segment that has a (potentially weak) match to activity during the previous time step (lines 33-36).

```
42. for c, i in cells
43.     for s in segments(c, i)
44.         if segmentActive(s, t, activeState) then
45.             predictiveState(c, i, t) = 1
46.
47.             activeUpdate = getSegmentActiveSynapses (c, i, s, t, false)
48.             segmentUpdateList.add(activeUpdate)
49.
50.             predSegment = getBestMatchingSegment(c, i, t-1)
51.             predUpdate = getSegmentActiveSynapses(
52.                                 c, i, predSegment, t-1, true)
53.             segmentUpdateList.add(predUpdate)
```

Phase 3

The third and last phase actually carries out learning. In this phase segment updates that have been queued up are actually implemented once we get feed-forward input and the cell is chosen as a learning cell (lines 38-40). Otherwise, if the cell ever stops predicting for any reason, we negatively reinforce the segments (lines 41-43).

```
54. for c, i in cells
55.     if learnState(s, i, t) == 1 then
56.         adaptSegments (segmentUpdateList(c, i), true)
57.         segmentUpdateList(c, i).delete()
58.     else if predictiveState(c, i, t) == 0 and predictiveState(c, i, t-1)==1 then
59.         adaptSegments (segmentUpdateList(c,i), false)
60.         segmentUpdateList(c, i).delete()
61.
```

Implementation details and terminology

In this section we describe some of the details of our temporal pooler implementation and terminology. Each cell is indexed using two numbers: a column index, c, and a cell index, i. Cells maintain a list of dendrite segments, where each segment contains a list of synapses plus a permanence value for each synapse. Changes to a cell's synapses are marked as temporary until the cell becomes active from feed forward input. These temporary changes are maintained in segmentUpdateList. Each segment also maintains a boolean flag, sequenceSegment, indicating whether the segment predicts feed-forward input on the next time step.

The implementation of potential synapses is different from the implementation in the spatial pooler. In the spatial pooler, the complete list of potential synapses is represented as an explicit list. In the temporal pooler, each segment can have its own (possibly large) list of potential synapses. In practice maintaining a long list for each segment is computationally expensive and memory intensive. Therefore in the temporal pooler, we randomly add active synapses to each segment during learning (controlled by the parameter newSynapseCount). This optimization has a similar effect to maintaining the full list of potential synapses, but the list per segment is far smaller while still maintaining the possibility of learning new temporal patterns.

The pseudocode also uses a small state machine to keep track of the cell states at different time steps. We maintain three different states for each cell. The arrays activeState and predictiveState keep track of the active and predictive states of each cell at each time step. The array learnState determines which cell outputs are used during learning. When an input is unexpected, all the cells in a particular column become active in the same time step. Only one of these cells (the cell that best matches the input) has its learnState turned on. We only add synapses from cells that have learnState set to one (this avoids overrepresenting a fully active column in dendritic segments).

The following data structures are used in the temporal pooler pseudocode:

cell(c,i)	A list of all cells, indexed by *i* and *c*.
cellsPerColumn	Number of cells in each column.
activeColumns(*t*)	List of column indices that are winners due to bottom-up input (this is the output of the spatial pooler).
activeState(c, i, t)	A boolean vector with one number per cell. It represents the active state of the column c cell i at time t given the current feed-forward input and the past temporal context. activeState(c, i, t) is the contribution from column c cell i at time t. If 1, the cell has current feed-forward input as well as an appropriate temporal context.
predictiveState(c, i, t)	A boolean vector with one number per cell. It represents the prediction of the column c cell i at time t, given the bottom-up activity of other columns and the past temporal context. predictiveState(c, i, t) is the contribution of column c cell i at time t. If 1, the cell is predicting feed-forward input in the current temporal context.
learnState(c, i, t)	A boolean indicating whether cell i in column c is chosen as the cell to learn on.
activationThreshold	Activation threshold for a segment. If the number of active connected synapses in a segment is greater than activationThreshold, the segment is said to be active.
learningRadius	The area around a temporal pooler cell from which it can get lateral connections.
initialPerm	Initial permanence value for a synapse.
connectedPerm	If the permanence value for a synapse is greater than this value, it is said to be connected.
minThreshold	Minimum segment activity for learning.
newSynapseCount	The maximum number of synapses added to a segment during learning.
permanenceInc	Amount permanence values of synapses are incremented when activity-based learning occurs.
permanenceDec	Amount permanence values of synapses are decremented when activity-based learning occurs.

segmentUpdate	Data structure holding three pieces of information required to update a given segment: a) segment index (-1 if it's a new segment), b) a list of existing active synapses, and c) a flag indicating whether this segment should be marked as a sequence segment (defaults to **false**).
segmentUpdateList	A list of segmentUpdate structures. segmentUpdateList(c,i) is the list of changes for cell i in column c.

The following supporting routines are used in the above code:

segmentActive(s, t, state)

This routine returns **true** if the number of connected synapses on segment s that are active due to the given state at time t is greater than activationThreshold. The parameter state can be activeState, or learnState.

getActiveSegment(c, i, t, state)

For the given column c cell i, return a segment index such that segmentActive(s,t, state) is **true**. If multiple segments are active, sequence segments are given preference. Otherwise, segments with most activity are given preference.

getBestMatchingSegment(c, i, t)

For the given column c cell i at time t, find the segment with the largest number of active synapses. This routine is aggressive in finding the best match. The permanence value of synapses is allowed to be below connectedPerm. The number of active synapses is allowed to be below activationThreshold, but must be above minThreshold. The routine returns the segment index. If no segments are found, then an index of -1 is returned.

getBestMatchingCell(c)

For the given column, return the cell with the best matching segment (as defined above). If no cell has a matching segment, then return the cell with the fewest number of segments.

getSegmentActiveSynapses(c, i, t, s, newSynapses= **false**)

Return a segmentUpdate data structure containing a list of proposed changes to segment s. Let activeSynapses be the list of active synapses where the originating cells have their activeState output = 1 at time step t. (This list is empty if s = -1 since the segment doesn't exist.) newSynapses is an optional argument that defaults to **false**. If newSynapses is **true**, then newSynapseCount - count(activeSynapses) synapses are added to activeSynapses. These synapses are randomly chosen from the set of cells that have learnState output = 1 at time step t.

adaptSegments(segmentList, positiveReinforcement)

This function iterates through a list of segmentUpdate's and reinforces each segment. For each segmentUpdate element, the following changes are performed. If positiveReinforcement is **true** then synapses on the active list get their permanence counts incremented by permanenceInc. All other synapses get their permanence counts decremented by permanenceDec. If positiveReinforcement is **false**, then synapses on the active list get their permanence counts decremented by permanenceDec. After this step, any synapses in segmentUpdate that do yet exist get added with a permanence count of initialPerm.

Glossary

	Notes: Definitions here capture how terms are used in this document, and may have other meanings in general use. Capitalized terms refer to other defined terms in this glossary.
Active State	a state in which Cells are active due to Feed-Forward input
Bottom-Up	synonym to Feed-Forward
Cells	HTM equivalent of a Neuron *Cells are organized into columns in HTM regions.*
Coincident Activity	two or more Cells are active at the same time
Column	a group of one or more Cells that function as a unit in an HTM Region *Cells within a column represent the same feed-forward input, but in different contexts.*
Dendrite Segment	a unit of integration of Synapses associated with Cells and Columns *HTMs have two different types of dendrite segments. One is associated with lateral connections to a cell. When the number of active synapses on the dendrite segment exceeds a threshold, the associated cell enters the predictive state. The other is associated with feed-forward connections to a column. The number of active synapses is summed to generate the feed-forward activation of a column.*
Desired Density	desired percentage of Columns active due to Feed-Forward input to a Region *The percentage only applies within a radius that varies based on the fan-out of feed-forward inputs. It is "desired" because the percentage varies some based on the particular input.*

Feed-Forward	moving in a direction away from an input, or from a lower Level to a higher Level in a Hierarchy (sometimes called Bottom-Up)
Feedback	moving in a direction towards an input, or from a higher Level to a lower level in a Hierarchy (sometimes called Top-Down)
First Order Prediction	a prediction based only on the current input and not on the prior inputs – compare to Variable Order Prediction
Hierarchical Temporal Memory (HTM)	a technology that replicates some of the structural and algorithmic functions of the neocortex
Hierarchy	a network of connected elements where the connections between the elements are uniquely identified as Feed-Forward or Feedback
HTM Cortical Learning Algorithms	the suite of functions for Spatial Pooling, Temporal Pooling, and learning and forgetting that comprise an HTM Region, also referred to as HTM Learning Algorithms
HTM Network	a Hierarchy of HTM Regions
HTM Region	the main unit of memory and Prediction in an HTM *An HTM region is comprised of a layer of highly interconnected cells arranged in columns. An HTM region today has a single layer of cells, whereas in the neocortex (and ultimately in HTM), a region will have multiple layers of cells. When referred to in the context of it's position in a hierarchy, a region may be referred to as a level.*
Inference	recognizing a spatial and temporal input pattern as similar to previously learned patterns
Inhibition Radius	defines the area around a Column that it actively inhibits
Lateral Connections	connections between Cells within the same Region
Level	an HTM Region in the context of the Hierarchy

Neuron	an information processing Cell in the brain *In this document, we use the word neuron specifically when referring to biological cells, and "cell" when referring to the HTM unit of computation.*
Permanence	a scalar value which indicates the connection state of a Potential Synapse *A permanence value below a threshold indicates the synapse is not formed. A permanence value above the threshold indicates the synapse is valid. Learning in an HTM region is accomplished by modifying permanence values of potential synapses.*
Potential Synapse	the subset of all Cells that could potentially form Synapses with a particular Dendrite Segment *Only a subset of potential synapses will be valid synapses at any time based on their permanence value.*
Prediction	activating Cells (into a predictive state) that will likely become active in the near future due to Feed-Forward input *An HTM region often predicts many possible future inputs at the same time.*
Receptive Field	the set of inputs to which a Column or Cell is connected *If the input to an HTM region is organized as a 2D array of bits, then the receptive field can be expressed as a radius within the input space.*
Sensor	a source of inputs for an HTM Network
Sparse Distributed Representation	representation comprised of many bits in which a small percentage are active and where no single bit is sufficient to convey meaning

Spatial Pooling	the process of forming a sparse distributed representation of an input *One of the properties of spatial pooling is that overlapping input patterns map to the same sparse distributed representation.*
Sub-Sampling	recognizing a large distributed pattern by matching only a small subset of the active bits in the large pattern
Synapse	connection between Cells formed while learning
Temporal Pooling	the process of forming a representation of a sequence of input patterns where the resulting representation is more stable than the input
Top-Down	synonym for Feedback
Variable Order Prediction	a prediction based on varying amounts of prior context – compare to First Order Prediction *It is called "variable" because the memory to maintain prior context is allocated as needed. Thus a memory system that implements variable order prediction can use context going way back in time without requiring exponential amounts of memory.*