BIOLOGICAL AND MACHINE INTELLIGENCE

(BAMI)

A digital book that documents Hierarchical Temporal Memory (HTM)

December 10, 2019

For more details on the use of Numenta's software and intellectual property, including the ideas contained in this book, see http://numenta.com/business-strategy-and-ip/.

Biological and Machine Intelligence (BAMI) is a digital book authored by Numenta researchers and engineers in 2016. It was created to document our theoretical framework for both biological and machine intelligence. Since the creation of BaMI, both the framework and our terminology have evolved. Notably, *HTM Theory* has changed to *The Thousand Brains Theory of Intelligence*.

While BAMI is not a complete book, it covers many of the fundamental concepts of our theory and contains detailed algorithms as of 2016. As we update this reference material in 2019, we note that the work documented in 2016 continues to be valid, but the theory has been substantially advanced. We have not yet written new BAMI chapters to describe the newer work, but instead include it here by referencing scientific papers that cover the new material.

We welcome your feedback and comments and may make revisions on an occasional basis.

BAMI Book Sections in this Document

Citing the Book

This release of Biological and Machine Intelligence is not close to being complete, so the book is not formally "published". However, we encourage you to cite this book in your own work by using one of these formats:

End Reference

```
Hawkins, J. et al. 2016-2020. Biological and Machine Intelligence. Release 0.4. Accessed athttps://numenta.com/resources/biological-and-machine-intelligence/.
```

Bibtex

```
@unpublished{Hawkins-et-al-2016-Book, title={Biological and Machine Intelligence (BAMI)}, author={Hawkins, J. and Ahmad, S. and Purdy, S. and Lavin, A.}, note={Initial online release 0.4}, url={https://numenta.com/resources/biological-and-machine-intelligence/}, year={2016} }
```

Note that some of the material in BAMI has been formally published; you can look at these papers to get the appropriate citations.

Introduction

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	J. Hawkins
0.41	Nov 25, 2019	Added information on Thousand Brains Theory of Intelligence to reflect thinking in 2019	C. Maver

Biological and Machine Intelligence: Introduction

The 21st century is a watershed in human evolution. We are solving the mystery of how the brain works and starting to build machines that work on the same principles as the brain. We are in the era of machine intelligence, which enables an explosion of beneficial applications and scientific advances.

Most people intuitively see the value in understanding how the human brain works. It is easy to see how brain theory could lead to the cure and prevention of mental disease or how it could lead to better methods for educating our children. These practical benefits justify the substantial efforts underway to reverse engineer the brain. However, the benefits go beyond the near-term and the practical. The human brain defines our species. In most aspects we are an unremarkable species, but our brain is unique. The large size of our brain, and its unique design, is the reason humans are the most successful species on our planet. Indeed, the human brain is the only thing we know of in the universe that can create and share knowledge. Our brains are capable of discovering the past, foreseeing the future, and unravelling the mysteries of the present. Therefore, if we want to understand who we are, if we want to expand our knowledge of the universe, and if we want to explore new frontiers, we need to have a clear understanding of how we know, how we learn, and how to build intelligent machines to help us acquire more knowledge. The ultimate promise of brain theory and machine intelligence is the acquisition and dissemination of new knowledge. Along the way there will be innumerous benefits to society. The beneficial impact of machine intelligence in our daily lives will equal and ultimately exceed that of programmable computers.

But exactly how will intelligent machines work and what will they do? If you suggest to a lay person that the way to build intelligent machines is to first understand how the human brain works and then build machines that work on the same principles as the brain, they will typically say, “That makes sense”. However, if you suggest this same path to artificial intelligence ("AI") and machine learning scientists, many will disagree. The most common rejoinder you hear is “airplanes don’t flap their wings”, suggesting that it doesn’t matter how brains work, or worse, that studying the brain will lead you down the wrong path, like building a plane that flaps its wings.

This analogy is both misleading and a misunderstanding of history. The Wright brothers and other successful pioneers of aviation understood the difference between the principles of flight and the need for propulsion. Bird wings and airplane wings work on the same aerodynamic principles, and those principles had to be understood before the Wright brothers could build an airplane. Indeed, they studied how birds glided and tested wing shapes in wind tunnels to learn the principles of lift. Wing flapping is different; it is a means of propulsion, and the specific method used for propulsion is less important when it comes to building flying machines. In an analogous fashion, we need to understand the principles of intelligence before we can build intelligent machines. Given that the only examples we have of intelligent systems are brains, and the principles of intelligence are not obvious, we must study brains to learn from them. However, like airplanes and birds, we don’t need to do everything the brain does, nor do we need to implement the principles of intelligence in the same way as the brain. We have a vast array of resources in software and silicon to create intelligent machines in novel and exciting ways. The goal of building intelligent machines is not to replicate human behavior, nor to build a brain, nor to create machines to do what humans do. The goal of building intelligent machines is to create machines that work on the same principles as the brain—machines that are able to learn, discover, and adapt in ways that computers can’t and brains can.

Consequently, the machine intelligence principles we describe in this book are derived from studying the brain. We use neuroscience terms to describe most of the principles, and we describe how these principles are implemented in the brain. The principles of intelligence can be understood by themselves, without referencing the brain, but for the foreseeable future it is easiest

to understand these principles in the context of the brain because the brain continues to offer suggestions and constraints on the solutions to many open issues.

This approach to machine intelligence is different than that taken by classic AI and artificial neural networks. Despite the fact that many AI pioneers are embracing brain-based approaches, most AI technologists' attempts to build intelligent machines involve encoding rules and knowledge in software and human-designed data structures. This AI approach has had many successes solving specific problems but has not offered a generalized approach to machine intelligence and, for the most part, has not addressed the question of how machines can learn. Artificial neural networks (ANNs) are learning systems built using networks of simple processing elements. In recent years ANNs, often called "deep learning networks", have succeeded in solving many classification problems. However, despite the word "neural", most ANNs are based on neuron models and network architectures that are incompatible with real biological tissue. More importantly, ANNs, by deviating from known brain principles, don't provide an obvious path to building truly intelligent machines.

Classic AI and ANNs generally are designed to solve specific types of problems rather than proposing a general theory of intelligence. In contrast, we know that brains use common principles for vision, hearing, touch, language, and behavior. This remarkable fact was first proposed in 1979 by Vernon Mountcastle. He said there is nothing visual about visual cortex and nothing auditory about auditory cortex. Every region of the neocortex performs the same basic operations. What makes the visual cortex visual is that it receives input from the eyes; what makes the auditory cortex auditory is that it receives input from the ears. From decades of neuroscience research, we now know this remarkable conjecture is true. Some of the consequences of this discovery are surprising. For example, neuroanatomy tells us that every region of the neocortex has both sensory and motor functions. Therefore, vision, hearing, and touch are integrated sensory-motor senses; we can't build systems that see and hear like humans do without incorporating movement of the eyes, body, and limbs.

The discovery that the neocortex uses common algorithms for everything it does is both elegant and fortuitous. It tells us that to understand how the neocortex works, we must seek solutions that are universal in that they apply to every sensory modality and capability of the neocortex. To think of vision as a "vision problem" is misleading. Instead we should think about vision as a "sensory motor problem" and ask how vision is the same as hearing, touch or language. Once we understand the common cortical principles, we can apply them to any sensory and behavioral systems, even those that have no biological counterpart. The theory and methods described in this book were derived with this idea in mind. Whether we build a system that sees using light or a system that "sees" using radar or a system that directly senses GPS coordinates, the underlying learning methods and algorithms will be the same.

Today, having made several important discoveries about how the neocortex works, we can build practical systems that solve valuable problems. Of course, there are still some things we don't understand about the brain and the neocortex, but we have an overall theory that can be tested. The theory includes key principles such as: how neurons make predictions, the role of dendritic spikes in cortical processing, how cortical layers learn sequences, and how cortical columns learn to model objects through movement. This book reflects those key principles.

The Thousand Brains Theory of Intelligence and Hierarchical Temporal Memory

When we wrote BaMI in 2016, we used the term HTM to mean two things: it was the name we used to describe the overall theory of how the neocortex functions. It was also the name we used to describe algorithmic components of the theory. In late 2018 we started referring to the theoretical framework for biological and machine intelligence as The Thousand Brains Theory of Intelligence. Hierarchical Temporal Memory, or HTM, now describes only the algorithmic components of that theory and related technical resources we've produced.

The Thousand Brains Theory of Intelligence is based on a discovery outlined in a 2019 peer-reviewed paper[1] that described how the brain does not learn only one model of an object or concept. Instead it builds many models, using different inputs from different sensors, and the models vote together to reach a consensus on what they're sensing. The consensus vote is what we perceive. It's as if your brain is actually thousands of brains working simultaneously.

Although the Thousand Brains Theory is a biologically constrained theory, and is perhaps the most biologically realistic theory of how the neocortex works, it does not attempt to include all biological details. Therefore the HTM algorithms do not include all biological details. For example, the biological neocortex exhibits several types of rhythmic behavior in the firing of ensembles of neurons. There is no doubt that these rhythms are essential for biological brains. But HTM algorithms do not include these rhythms because we don't believe they play an information-theoretic role. Our best guess is that these rhythms are needed in biological brains to synchronize action potentials, but we don't have this issue in software and hardware implementations of HTM. If in the future we find that rhythms are essential for intelligence, and not just biological brains, then we would modify HTM algorithms to

[1] A Framework for Intelligence and Cortical Function Based on Grid Cells in the Neocortex

include them. There are many biological details that similarly are not part of HTM. Every feature included in HTM is there because we have an information-theoretical need that is met by that feature.

The Thousand Brains Theory is not a theory of an entire brain; it only covers the neocortex and its interactions with some closely related structures such as the thalamus and hippocampus. The neocortex is where most of what we think of as intelligence resides but it is not in charge of emotions, homeostasis, and basic behaviors. Other, evolutionarily older, parts of the brain perform these functions. These older parts of the brain have been under evolutionary pressure for much longer time, and although they consist of neurons, they are heterogeneous in architecture and function. We are not interested in emulating entire brains or in making machines that are human-like, with human-like emotions and desires. Therefore intelligent machines, as we define them, are not likely to pass the Turing test or be like the humanoid robots seen in science fiction. This distinction does not suggest that intelligent machines will be of limited utility. Many will be simple, tirelessly sifting through vast amounts of data looking for unusual patterns. Others will be fantastically fast and smart, able to explore domains that humans are not well suited for. The variety we will see in intelligent machines will be similar to the variety we see in programmable computers. Some computers are tiny and embedded in cars and appliances, and others occupy entire buildings or are distributed across continents. Intelligent machines will have a similar diversity of size, speed, and applications, but instead of being programmed they will learn.

The Thousand Brains Theory cannot be expressed succinctly in one or a few mathematical equations. HTM principles work together to produce perception and behavior. In this regard, HTMs are like computers. Computers can't be described purely mathematically. We can understand how they work, we can simulate them, and subsets of computer science can be described in formal mathematics, but ultimately we have to build them and test them empirically to characterize their performance. Similarly, some parts of HTM can be analyzed mathematically. For example, the chapter in this book on sparse distributed representations is mostly about the mathematical properties of sparse representations. But other parts of the theory are less amenable to formalism. If you are looking for a succinct mathematical expression of intelligence, you won't find it. In this way, brain theory is more like genetic theory and less like physics.

What is Intelligence?

Historically, intelligence has been defined in behavioral terms. For example, if a system can play chess, or drive a car, or answer questions from a human, then it is exhibiting intelligence. The Turing Test is the most famous example of this line of thinking. We believe this approach to defining intelligence fails on two accounts. First, there are many examples of intelligence in the biological world that differ from human intelligence and would fail most behavioral tests. For example, dolphins, monkeys, and humans are all intelligent, yet only one of these species can play chess or drive a car. Similarly, intelligent machines will span a range of capabilities from mouse-like to super-human and, more importantly, we will apply intelligent machines to problems that have no counterpart in the biological world. Focusing on human-like performance is limiting.

The second reason we reject behavior-based definitions of intelligence is that they don't capture the incredible flexibility of the neocortex. The neocortex uses the same algorithms for all that it does, giving it flexibility that has enabled humans to be so successful. Humans can learn to perform a vast number of tasks that have no evolutionary precedent because our brains use learning algorithms that can be applied to almost any task. The way the neocortex sees is the same as the way it hears or feels. In humans, this universal method creates language, science, engineering, and art. When we define intelligence as solving specific tasks, such as playing chess, we tend to create solutions that also are specific. The program that can win a chess game cannot learn to drive. It is the flexibility of biological intelligence that we need to understand and embed in our intelligent machines, not the ability to solve a particular task. Another benefit of focusing on flexibility is network effects. The neocortex may not always be best at solving any particular problem, but it is very good at solving a huge array of problems. Software engineers, hardware engineers, and application engineers naturally gravitate towards the most universal solutions. As more investment is focused on universal solutions, they will advance faster and get better relative to other more dedicated methods. Network effects have fostered adoption many times in the technology world; this dynamic will unfold in the field of machine intelligence, too.

Therefore we define the intelligence of a system by the degree to which it exhibits flexibility: flexibility in learning and flexibility in behavior. Since the neocortex is the most flexible learning system we know of, we measure the intelligence of a system by how many of the neocortical principles that system includes. This book is an attempt to enumerate and understand these neocortical principles. Any system that includes all the principles we cover in this book will exhibit cortical-like flexibility, and therefore cortical-like intelligence. By making systems larger or smaller and by applying them to different sensors and embodiments, we can create intelligent machines of incredible variety. Many of these systems will be much smaller than a human neocortex and some will be much larger in terms of memory size, but they will all be intelligent.

About this Book

The structure of this book may be different from those you have read in the past. Some of the chapters cover key principles of the Thousand Brains Theory in great detail. Later chapters instead point to published papers and additional resources. Our hope is that

we have covered aspects of the theory that are best understood. Some chapters may be published in draft form, whereas others will be more polished.

Second, the book is intended for a technical but diverse audience. Neuroscientists should find the book helpful as it provides a theoretical framework to interpret many biological details and guide experiments. Computer scientists can use the material in the book to develop machine intelligence hardware, software, and applications based on neuroscience principles. Anyone with a deep interest in how brains work or machine intelligence will hopefully find the book to be the best source for these topics. Finally, we hope that academics and students will find this material to be a comprehensive introduction to an emerging and important field that offers opportunities for future research and study.

The structure of the chapters in this book varies depending on the topic. Some chapters are overview in nature. Some chapters include mathematical formulations and problem sets to exercise the reader's knowledge. Some chapters include pseudo-code. Key citations will be noted, but we do not attempt to have a comprehensive set of citations to all work done in the field. As such, we gratefully acknowledge the many pioneers whose work we have built upon who are not explicitly mentioned.

We are now ready to jump into the details of biological and machine intelligence.

HTM Overview

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	J. Hawkins
0.41	May 23, 2016	Fixed references to the time frame of human neocortex development	
0.42	June 22, 2016	Added figure 1	J. Hawkins
0.43	Oct 13, 2016	Corrected neocortex size reference	
0.44	Nov 25, 2019	Updated terminology to reflect current thinking on Thousand Brains Theory of Intelligence	C. Maver

Hierarchical Temporal Memory: Overview

The Thousand Brains Theory of Intelligence is a biological theory, meaning it is derived from neuroanatomy and neurophysiology and explains how the biological neocortex works. HTM principles are the technical details and algorithms that comprise the theory. Much of this book focuses on those HTM principles. We sometimes say HTM is "biologically constrained," as opposed to "biologically inspired," which is a term often used in machine learning. The biological details of the neocortex must be compatible with the theory, and the theory can't rely on principles that can't possibly be implemented in biological tissue. For example, consider the pyramidal neuron, the most common type of neuron in the neocortex. Pyramidal neurons have tree-like extensions called dendrites that connect via thousands of synapses. Neuroscientists know that the dendrites are active processing units, and that communication through the synapses is a dynamic, inherently stochastic process (Poirazi and Mel, 2001). The pyramidal neuron is the core information processing element of the neocortex, and synapses are the substrate of memory. Therefore, to understand how the neocortex works we need a theory that accommodates the essential features of neurons and synapses. Artificial Neural Networks (ANNs) usually model neurons with no dendrites and few highly precise synapses, features which are incompatible with real neurons. This type of artificial neuron can't be reconciled with biological neurons and is therefore unlikely to lead to networks that work on the same principles as the brain. This observation doesn't mean ANNs aren't useful, only that they don't work on the same principles as biological neural networks. As you will see, our theory explains why neurons have thousands of synapses and active dendrites. We believe these and many other biological features are essential for an intelligent system and can't be ignored.

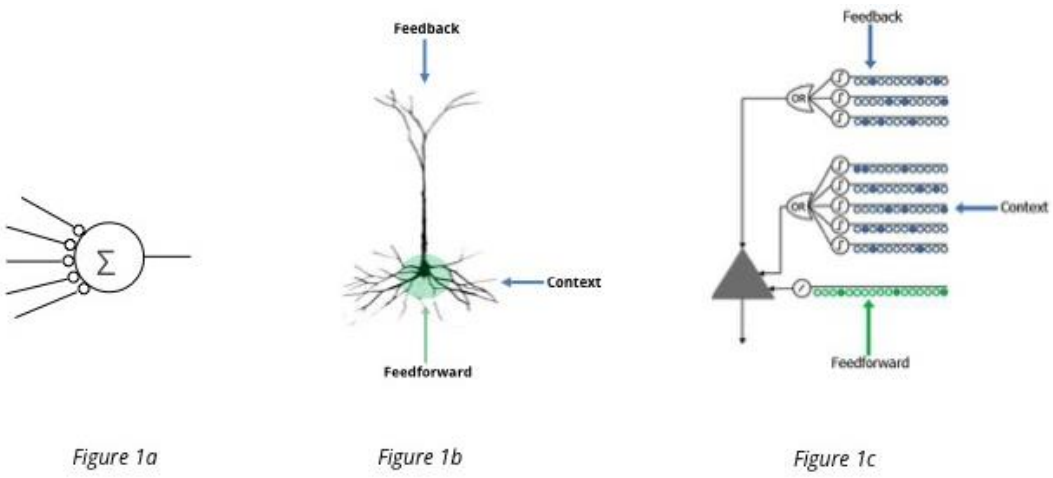


Figure 1a Figure 1b Figure 1c

Figure 1 Biological and artificial neurons. *Figure 1a* shows an artificial neuron typically used in machine learning and artificial neural networks. Often called a "point neuron" this form of artificial neuron has relatively few synapses and no dendrites. Learning in a point neuron occurs by changing the "weight" of the synapses which are represented by a scalar value that can take a positive or negative value. A point neuron calculates a weighted sum of its inputs which is applied to a non-linear function to determine the output value of the neuron. *Figure 1b* shows a pyramidal neuron which is the most common type of neuron in the neocortex. Biological neurons have thousands of synapses arranged along dendrites. Dendrites are active processing elements allowing the neuron to recognize hundreds of unique patterns. Biological synapses are partly stochastic and therefore are low precision. Learning in a biological neuron is mostly due to the formation of new synapses and the removal of unused synapses. Pyramidal neurons have multiple synaptic integration zones that receive input from different sources and have differing effects on the cell. *Figure 1c* shows an HTM artificial neuron. Similar to a pyramidal neuron it has thousands of synapses arranged on active dendrites. It recognizes hundreds of patterns in multiple integration zones. The HTM neuron uses binary synapses and learns by modeling the growth of new synapses and the decay of unused synapses. HTM neurons don't attempt to model all aspects of biological neurons, only those that are essential for information theoretic aspects of the neocortex.

Although we want to understand how biological brains work, we don't have to adhere to all the biological details. Once we understand how real neurons work and how biological networks of neurons lead to memory and behavior, we might decide to implement them in software or in hardware in a way that differs from the biology in detail, but not in principle. But we shouldn't do that before we understand how biological neural systems work. People often ask, "How do you know which biological details matter and which don't?" The answer is: once you know how the biology works, you can decide which biological details to include in your models and which to leave out, but you will know what you are giving up, if anything, if your software model leaves out a particular biological feature. Human brains are just one implementation of intelligence; yet today, humans are the only things everyone agrees are intelligent. Our challenge is to separate aspects of brains and neurons that are essential for intelligence from those aspects that are artifacts of the brain's particular implementation of intelligence principles. Our goal isn't to recreate a brain, but to understand how brains work in sufficient detail so that we can test the theory biologically and also build systems that, although not identical to brains, work on the same principles.

Sometime in the future designers of intelligent machines may not care about brains and the details of how brains implement the principles of intelligence. The field of machine intelligence may by then be so advanced that it has departed from its biological origin. But we aren't there yet. Today we still have much to learn from biological brains and therefore to understand HTM principles and to build intelligent machines, it is necessary to know neuroscience terms and the basics of the brain's design.

The remainder of this chapter introduces some of the key concepts of HTM. We will describe an aspect of the neocortex and then relate that biological feature to one or more principles of HTM. In-depth descriptions and technical details of the HTM principles are provided in subsequent chapters.

You'll find information on the representation format used by the neocortex ("sparse distributed representations" or SDRs), the mathematical and semantic operations that are enabled by this representation format, and how neurons throughout the neocortex learn sequences and make predictions, which is the core component of all inference and behavior. We also understand how knowledge is stored by the formation of sets of new synapses on the dendrites of neurons. These are basic elements of biological intelligence analogous to how random access memory, busses, and instruction sets are basic elements of computers. Once you understand these basic elements, you can combine them in different ways to create full systems.

Biological Observation: The Structure of the Neocortex

The human brain comprises several components such as the brain stem, the basal ganglia, and the cerebellum. These organs are loosely stacked on top of the spinal cord. The neocortex is just one more component of the brain, but it dominates the human brain, occupying about 75% of its volume. The evolutionary history of the brain is reflected in its overall design. Simple animals, such as worms, have the equivalent of the spinal cord and nothing else. The spinal cord of a worm and a human receives sensory input and generates useful, albeit simple, behaviors. Over evolutionary time scales, new brain structures were added such as the brainstem and basal ganglia. Each addition did not replace what was there before. Typically the new brain structure received input from the older parts of the brain, and the new structure's output led to new behaviors by controlling the older brain regions. The addition of each new brain structure had to incrementally improve the animal's behaviors. Because of this evolutionary path, the entire brain is physically and logically a hierarchy of brain regions.

Figure 2 a) real brain b) logical hierarchy *(placeholder)*

The neocortex is the most recent addition to our brain. All mammals, and only mammals, have a neocortex. The neocortex first appeared about 200 million years ago in the early small mammals that emerged from their reptilian ancestors during the transition of the Triassic/Jurassic periods. The modern human neocortex separated from those of monkeys in terms of size and complexity about 25 million years ago (Rakic, 2009). The human neocortex continued to evolve to be bigger and bigger, reaching its current size in humans between 800,000 and 200,000 years ago[2]. In humans, the neocortex is a sheet of neural tissue about the size of a large dinner napkin (2,500 square centimeters) in area and 2.5mm thick. It lies just under the skull and wraps around the other parts of the brain. (From here on, the word "neocortex" will refer to the human neocortex. References to the neocortex of other mammals will be explicit.) The neocortex is heavily folded to fit in the skull but this isn't important to how it works, so we will always refer to it and illustrate it as a flat sheet. The human neocortex is large both in absolute terms and also relative to the size of our body compared to other mammals. We are an intelligent species mostly because of the size of our neocortex.

The most remarkable aspect of the neocortex is its homogeneity. The types of cells and their patterns of connectivity are nearly identical no matter what part of the neocortex you look at. This fact is largely true across species as well. Sections of human, rat, and monkey neocortex look remarkably the same. The primary difference between the neocortex of different animals is the size of the neocortical sheet. Many pieces of evidence suggest that the human neocortex got large by replicating a basic element over and over. This observation led to the 1978 conjecture by Vernon Mountcastle that every part of the neocortex must be doing the same thing. So even though some parts of the neocortex process vision, some process hearing, and other parts create language, at a fundamental level these are all variations of the same problem, and are solved by the same neural algorithms. Mountcastle argued that the vision regions of the neocortex are vision regions because they receive input from the eyes and not because they have special vision neurons or vision algorithms (Mountcastle, 1978). This discovery is incredibly important and is supported by multiple lines of evidence.

Even though the neocortex is largely homogenous, some neuroscientists are quick to point out the differences between neocortical regions. One region may have more of a certain cell type, another region has extra layers, and other regions may exhibit variations in connectivity patterns. But there is no question that neocortical regions are remarkably similar and that the variations are relatively minor. The debate is only about how critical the variations are in terms of functionality.

The neocortical sheet is divided into dozens of regions situated next to each other. Looking at a neocortex you would not see any regions or demarcations. The regions are defined by connectivity. Regions pass information to each other by sending bundles of nerve fibers into the white matter just below the neocortex. The nerve fibers reenter at another neocortical region. The connections between regions define a logical hierarchy. Information from a sensory organ is processed by one region, which passes its output to another region, which passes its output to yet another region, etc. The number of regions and their connectivity is determined by our genes and is the same for all members of a species. So, as far as we know, the hierarchical organization of each human's neocortex is the same, but our hierarchy differs from the hierarchy of a dog or a whale. The actual hierarchy for some species has been mapped in detail (Zingg, 2014). They are complicated, not like a simple flow chart. There are parallel paths up the hierarchy and information often skips levels and goes sideways between parallel paths. Despite this complexity the hierarchical structure of the neocortex is well established.

We can now see the big picture of how the brain is organized. The entire brain is a hierarchy of brain regions, where each region interacts with a fully functional stack of evolutionarily older regions below it. For most of evolutionary history new brain regions, such as the spinal cord, brain stem, and basal ganglia, were heterogeneous, adding capabilities that were specific to particular senses and behaviors. This evolutionary process was slow. Starting with mammals, evolution discovered a way to extend the brain's hierarchy using new brain regions with a homogenous design, an algorithm that works with any type of sensor data and any type of behavior. This replication is the beauty of the neocortex. Once the universal neocortical algorithms were established, evolution

[2] http://humanorigins.si.edu/human-characteristics/brains

could extend the brain's hierarchy rapidly because it only had to replicate an existing structure. This explains how human brains evolved to become large so quickly.

Figure 3 a) brain with information flowing posterior to anterior b) logical hierarchical stack showing old brain regions and neocortical regions *(placeholder)*

Sensory information enters the human neocortex in regions that are in the rear and side of the head. As information moves up the hierarchy it eventually passes into regions in the front half of the neocortex. Some of the regions at the very top of the neocortical hierarchy, in the frontal lobes and also the hippocampus, have unique properties such as the ability for short term memory, which allows you to keep a phone number in your head for a few minutes. These regions also exhibit more heterogeneity, and some of them are older than the neocortex. The neocortex in some sense was inserted near the top of the old brain's hierarchical stack. Therefore, we first try to understand the homogenous regions that are near the bottom of the neocortical hierarchy. In other words, we first need to understand how the neocortex builds a basic model of the world from sensory data and how it generates basic behaviors.

HTM Principle: Common Algorithms

Our theory focuses on the common properties across the neocortex. We strive not to understand vision or hearing or robotics as separate problems, but to understand how these capabilities are fundamentally all the same, and what set of algorithms can see AND hear AND generate behavior. Initially, this general approach makes our task seem harder, but ultimately it is liberating. When we describe or study a particular HTM learning algorithm we often will start with a particular problem, such as vision, to understand or test the algorithm. But we then ask how the exact same algorithm would work for a different problem such as understanding language. This process leads to realizations that might not at first be obvious, such as vision being a primarily temporal inference problem, meaning the temporal order of patterns coming from the retina is as important in vision as is the temporal order of words in language. Once we understand the common algorithms of the neocortex, we can ask how evolution might have tweaked these algorithms to achieve even better performance on a particular problem. But our focus is to first understand the common algorithms that are manifest in all neocortical regions.

Biological Observation: Neurons are Sparsely Activated

The neocortex is made up of neurons. No one knows exactly how many neurons are in a human neocortex, but recent "primate scale up" methods put the estimate at 86 billion (Herculano-Houzel, 2012). The moment-to-moment state of the neocortex, some of which defines our perceptions and thoughts, is determined by which neurons are active at any point in time. An active neuron is one that is generating spikes, or action potentials. One of the most remarkable observations about the neocortex is that no matter where you look, the activity of neurons is sparse, meaning only a small percentage of them are rapidly spiking at any point in time. The sparsity might vary from less than one percent to several percent, but it is always sparse.

HTM Principle: Sparse Distributed Representations

The representations used in HTM are called Sparse Distributed Representations, or SDRs. SDRs are vectors with thousands of bits. At any point in time a small percentage of the bits are 1's and the rest are 0's. Our theory explains why it is important that there are always a minimum number of 1's distributed in the SDR, and also why the percentage of 1's must be low, typically less than 2%. The bits in an SDR correspond to the neurons in the neocortex.

SDRs have some essential and surprising properties. For comparison, consider the representations used in programmable computers. The meaning of a word in a computer is not inherent in the word itself. If you were shown 64 bits from a location in a computer's memory you couldn't say anything about what it represents. At one moment in the execution of the program the bits could represent one thing and at another moment they might represent something else, and in either case the meaning of the 64 bits can only be known by relying on knowledge not contained in the physical location of the bits themselves. With SDRs, the bits of the representation encode the semantic properties of the representation; the representation and its meaning are one and the same. Two SDRs that have 1 bits in the same location share a semantic property. The more 1 bits two SDRs share, the more semantically similar are the two representations. The SDR explains how brains make semantic generalizations; it is an inherent property of the representation method. Another example of a unique capability of sparse representations is that a set of neurons can simultaneously activate multiple representations without confusion. It is as if a location in computer memory could hold not just one value but twenty simultaneous values and not get confused! We call this unique characteristic the "union property" and it is used throughout HTM for such things as making multiple predictions at the same time.

The use of sparse distributed representations is a key component of HTM. We believe that all truly intelligent systems must use sparse distributed representations.

Biological Observation: The Inputs and Outputs of the Neocortex

As mentioned earlier, the neocortex appeared recently in evolutionary time. The other parts of the brain existed before the neocortex appeared. You can think of a human brain as consisting of a reptile brain (the old stuff) with a neocortex (literally "new layer") attached on top of it. The older parts of the brain still have the ability to sense the environment and to act. We humans still have a reptile inside of us. The neocortex is not a stand-alone system, it learns how to interact with and control the older brain areas to create novel and improved behaviors.

There are two basic inputs to the neocortex. One is data from the senses. As a general rule, sensory data is processed first in the sensory organs such as the retina, cochlea, and sensory cells in the skin and joints. It then goes to older brain regions that further process it and control basic behaviors. Somewhere along this path the neurons split their axons in two and send one branch to the neocortex. The sensory input to the neocortex is literally a copy of the sensory data that the old brain is getting.

The second input to the neocortex is a copy of motor commands being executed by the old parts of the brain. For example, walking is partially controlled by neurons in the brain stem and spinal cord. These neurons also split their axons in two, one branch generates behavior in the old brain and the other goes to the neocortex. Another example is eye movements, which are controlled by an older brain structure called the superior colliculus. The axons of superior colliculus neurons send a copy of their activity to the neocortex, letting the neocortex know what movement is about to happen. This motor integration is a nearly universal property of the brain. The neocortex is told what behaviors the rest of the brain is generating as well as what the sensors are sensing. Imagine what would happen if the neocortex wasn't informed that the body was moving in some way. If the neocortex didn't know the eyes were about to move, and how, then a change of pattern on the optic nerve would be perceived as the world moving. The fact that our perception is stable while the eyes move tells us the neocortex is relying on knowledge of eye movements. When you touch, hear, or see something, the neocortex needs to distinguish changes in sensation caused by your own movement from changes caused by movements in the world. The majority of changes on your sensors are the result of your own movements. This "sensory-motor" integration is the foundation of how most learning occurs in the neocortex. The neocortex uses behavior to learn the structure of the world.

Figure 4. showing sensory & motor command inputs to the neocortex (block diagram) *(placeholder)*

No matter what the sensory data represents - light, sound, touch or behavior - the patterns sent to the neocortex are constantly changing. The flowing nature of sensory data is perhaps most obvious with sound, but the eyes move several times a second, and to feel something we must move our fingers over objects and surfaces. Irrespective of sensory modality, input to the neocortex is like a movie, not a still image. The input patterns completely change typically several times a second. The changes in input are not something the neocortex has to work around, or ignore; instead, they are essential to how the neocortex works. The neocortex is memory of time-based patterns.

The primary outputs of the neocortex come from neurons that generate behavior. However, the neocortex never controls muscles directly; instead the neocortex sends its axons to the old brain regions that actually generate behavior. Thus the neocortex tries to control the old brain regions that in turn control muscles. For example, consider the simple behavior of breathing. Most of the time breathing is controlled completely by the brain stem, but the neocortex can learn to control the brain stem and therefore exhibit some control of breathing when desired.

A region of neocortex doesn't "know" what its inputs represent or what its output might do. A region doesn't even "know" where it is in the hierarchy of neocortical regions. A region accepts a stream of sensory data plus a stream of motor commands. From these inputs it learns of the changes in the inputs. The region will output a stream of motor commands, but it only knows how its output changes its input. The outputs of the neocortex are not pre-wired to do anything. The neocortex has to learn how to control behavior via associative linking.

HTM Principle: Sensory Encoders

Every HTM system needs the equivalent of sensory organs. We call these "encoders." An encoder takes some type of data–it could be a number, time, temperature, image, or GPS location–and turns it into a sparse distributed representation that can be digested by the HTM learning algorithms. Encoders are designed for specific data types, and often there are multiple ways an encoder can convert an input to an SDR, in the same way that there are variations of retinas in mammals. The HTM learning algorithms will work with any kind of sensory data as long as it is encoded into proper SDRs.

One of the exciting aspects of machine intelligence based on HTM principles is that we can create encoders for data types that have no biological counterpart. For example, we have created an encoder that accepts GPS coordinates and converts them to SDRs. This encoder allows an HTM system to directly sense movement through space. The HTM system can then classify movements, make predictions of future locations, and detect anomalies in movements. The ability to use non-human senses offers a hint of where intelligent machines might go. Instead of intelligent machines just being better at what humans do, they will be applied to problems where humans are poorly equipped to sense and to act.

HTM Principle: HTM Systems are Embedded Within Sensory-motor Systems

To create an intelligent system, the HTM learning algorithms need both sensory encoders and some type of behavioral framework. You might say that the HTM learning algorithms need a body. But the behaviors of the system do not need to be anything like the behaviors of a human or robot. Fundamentally, behavior is a means of moving a sensor to sample a different part of the world. For example, the behavior of an HTM system could be traversing links on the world-wide web or exploring files on a server.

It is possible to create HTM systems without behavior. If the sensory data naturally changes over time, then an HTM system can learn the patterns in the data, classify the patterns, detect anomalies, and make predictions of future values. The early work on HTM focuses on these kinds of problems, without a behavioral component. Ultimately, to realize the full potential of HTM, behavior needs to be incorporated fully.

HTM Principle: HTM Relies On Streaming Data and Sequence Memory

The HTM learning algorithms are designed to work with sensor and motor data that is constantly changing. Sensor input data may be changing naturally such as metrics from a server or the sounds of someone speaking. Alternately the input data may be changing because the sensor itself is moving such as moving the eyes while looking at a still picture. At the heart of HTM is a learning algorithm called Temporal Memory, or TM. As its name implies, Temporal Memory is a memory of sequences, it is a memory of transitions in a data stream. TM is used in both sensory inference and motor generation. Our theory postulates that every excitatory neuron in the neocortex is learning transitions of patterns and that the majority of synapses on every neuron are dedicated to learning these transitions. Temporal Memory is therefore the substrate upon which all neocortical functions are built. TM is probably the biggest difference between HTM and most other artificial neural networks. HTM starts with the assumption that everything the neocortex does is based on memory and recall of sequences of patterns.

HTM Principle: On-line Learning

HTM systems learn continuously, which is often referred to as “on-line learning”. With each change in the inputs the memory of the HTM system is updated. There are no batch learning data sets and no batch testing sets as is the norm for most machine learning algorithms. Sometimes people ask, “If there are no labeled training and test sets, how does the system know if it is working correctly and how can it correct its behavior?” HTM builds a predictive model of the world, which means that at every point in time the HTM-based system is predicting what it expects will happen next. The prediction is compared to what actually happens and forms the basis of learning. HTM systems try to minimize the error of their predictions.

Another advantage of continuous learning is that the system will constantly adapt if the patterns in the world change. For a biological organism this is essential to survival. Our theory is built on the assumption that intelligent machines need to continuously learn as well. However, there will likely be applications where we don’t want a system to learn continuously, but these are the exceptions, not the norm.

Conclusion

The biology of the neocortex informs our theory and the HTM algorithms. In the following chapters we discuss details and continue to draw parallels between HTM and the neocortex.

References

Crick, F. H.C. (1979) Thinking About the Brain. Scientific American September 1979, pp. 229, 230. Ch. 4 27

Poirazi, P. & Mel, B. W. (2001) Impact of active dendrites and structural plasticity on the memory capacity of neural tissue. Neuron, 2001 doi:10.1016/S0896-6273(01)00252-5

Rakic, P. (2009). Evolution of the neocortex: Perspective from developmental biology. Nature Reviews Neuroscience. Retrieved from http://www.ncbi.nlm.nih.gov/pmc/articles/PMC2913577/

Mountcastle, V. B. (1978) An Organizing Principle for Cerebral Function: The Unit Model and he Distributed System, in Gerald M. Edelman & Vernon V. Mountcastle, ed., 'The Mindful Brain' , MIT Press, Cambridge, MA , pp. 7-50

Zingg, B. (2014) Neural networks of the mouse neocortex. Cell, 2014 Feb 27;156(5):1096-111. doi: 10.1016/j.cell.2014.02.023

Herculano-Houzel, S. (2012). The remarkable, yet not extraordinary, human brain as a scaled-up primate brain and its associated cost. Proceedings of the National Academy of Sciences of the United States of America, 109 (Suppl 1), 10661–10668. http://doi.org/10.1073/pnas.1201895109

Sparse Distributed Representations

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	A. Lavin, S. Ahmad, J. Hawkins
0.41	Dec 21, 2016	Replaced figure 5 and made some clarifications.	S. Ahmad
0.57	Nov 19, 2021	Corrected equation 12	C. Lai
0.57	May 4, 2022	Corrected equation 9 and made some clarifications	S. Ahmad

Sparse Distributed Representations

In this chapter we introduce Sparse Distributed Representations (SDRs), the fundamental form of information representation in the brain, and in HTM systems. We talk about several interesting and useful mathematical properties of SDRs and then discuss how SDRs are used in the brain.

What is a Sparse Distributed Representation?

One of the most interesting challenges in AI is the problem of knowledge representation. Representing everyday facts and relationships in a form that computers can work with has proven to be difficult with traditional computer science methods. The basic problem is that our knowledge of the world is not divided into discrete facts with well-defined relationships. Almost everything we know has exceptions, and the relationships between concepts are too numerous and ill-defined to map onto traditional computer data structures.

Brains do not have this problem. They represent information using a method called Sparse Distributed Representations, or SDRs. SDRs and their mathematical properties are essential for biological intelligence. Everything the brain does and every principle described in this book is based on SDRs. SDRs are the language of the brain. The flexibility and creativity of human intelligence is inseparable from this representation method. Therefore, if we want intelligent machines to be similarly flexible and creative, they need to be based on the same representation method, SDRs.

An SDR consists of thousands of bits where at any point in time a small percentage of the bits are 1's and the rest are 0's. The bits in an SDR correspond to neurons in the brain, a 1 being a relatively active neuron and a 0 being a relatively inactive neuron. The most important property of SDRs is that each bit has meaning. Therefore, the set of active bits in any particular representation encodes the set of semantic attributes of what is being represented. The bits are not labeled (that is to say, no one assigns meanings to the bits), but rather, the semantic meanings of bits are learned. If two SDRs have active bits in the same locations, they share the semantic attributes represented by those bits. By determining the overlap between two SDRs (the equivalent bits that are 1 in both SDRs) we can immediately see how two representations are semantically similar and how they are semantically different. Because of this semantic overlap property, systems based on SDRs automatically generalize based on semantic similarity.

HTM theory defines how to create, store, and recall SDRs and sequences of SDRs. SDRs are not moved around in memory, like data in computers. Instead the set of active neurons, within a fixed population of neurons, changes over time. At one moment a set of neurons represents one thing; the next moment it represents something else. Within one set of neurons, an SDR at one point in time can associatively link to the next occurring SDR. In this way, sequences of SDRs are learned. Associative linking also occurs between different populations of cells (layer to layer or region to region). The meanings of the neuron encodings in one region are different than the meanings of neuron encodings in another region. In this way, an SDR in one modality, such as a sound, can associatively invoke an SDR in another modality, such as vision.

Any type of concept can be encoded in an SDR, including different types of sensory data, words, locations, and behaviors. This is why the neocortex is a universal learning machine. The individual regions of the neocortex operate on SDRs without "knowing" what the SDRs represent in the real world. HTM systems work the same way. As long as the inputs are in a proper SDR format, the HTM algorithms will work. In an HTM-based system, knowledge is inherent in the data, not in the algorithms.

To better understand the properties of SDRs it can be helpful to think about how information is typically represented in computers and the relative merits of SDRs versus the representation scheme used by computers. In computers, we represent information with bytes and words. For example, to represent information about a medical patient, a computer program might use one byte to store the patient's age and another byte to store the patient's gender. Data structures such as lists and trees are used to organize pieces of information that are related. This type of representation works well when the information we need to represent is well defined and limited in extent. However, AI researchers discovered that to make a computer intelligent, it needs to know a huge amount of knowledge, the structure of which is not well defined.

For example, what if we want our intelligent computer to know about cars? Think of all the things you know about cars. You know what they do, how to drive them, how to get in and out of them, how to clean them, ways they can fail, what the different controls do, what is found under the hood, how to change tires, etc. We know the shapes of cars and the sounds they make. If you just think about tires, you might recall different types of tires, different brands of tires, how they wear unevenly, the best way to rotate them, etc. The list of all the things you know about cars goes on and on.

Each piece of knowledge leads to other pieces of knowledge in an ever-expanding web of associations. For example, cars have doors, but other objects have doors too, such as houses, planes, mailboxes, and elevators. We intuitively know what is similar and different about all these doors and we can make predictions about new types of doors we have never seen before based on previous experience. We (our brains) find it easy to recall a vast number of facts and relationships via association. But when AI scientists try to encode this type of knowledge into a computer, they find it difficult.

In computers information is represented using words of 8, 32, or 64 bits. Every combination of 1's and 0's is used, from all 1's to all 0's, which is sometimes called a "dense" representation. An example of a dense representation is the ASCII code for letters of the alphabet. In ASCII the letter "m" is represented by:

01101101

Notice it is the combination of all eight bits that encodes "m", and the individual bits in this representation don't mean anything on their own. You can't say what the third bit means; the combination of all eight bits is required. Notice also that dense representations are brittle. For example, if you change just one bit in the ASCII code for "m" as follows:

01100101

you get the representation for an entirely different letter, "e". One wrong bit and the meaning changes completely.

There is nothing about 01101101 that tells you what it represents or what attributes it has. It is a purely arbitrary and abstract representation. The meaning of a dense representation must be stored elsewhere.

In contrast, as mentioned earlier, SDRs have thousands of bits. At every point in time a small percentage of the bits are 1's and the rest are 0's. An SDR may have 2,000 bits with 2% (or 40) being 1 and the rest 0, as visualized in Figure 1.

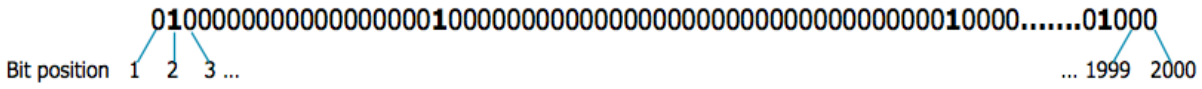


Figure 1: An SDR of 2000 bits, where only a few are ON (ones).

In an SDR, each bit has meaning. For example if we want to represent letters of the alphabet using SDRs, there may be bits representing whether the letter is a consonant or a vowel, bits representing how the letter sounds, bits representing where in the alphabet the letter appears, bits representing how the letter is drawn (i.e. open or closed shape, ascenders, descenders), etc. To represent a particular letter, we pick the 40 attributes that best describe that letter and make those bits 1. In practice, the meaning of each bit is learned rather than assigned; we are using letters and their attributes for illustration of the principles.

With SDRs the meaning of the thing being represented is encoded in the set of active bits. Therefore, if two different representations have a 1 in the same location we can be certain that the two representations share that attribute. Because the representations are sparse, two representations will not share an attribute by chance; a shared bit/attribute is always meaningful. As shown in Figure 2, simply comparing SDRs in this way tells us how any two objects are semantically similar and how they are different.

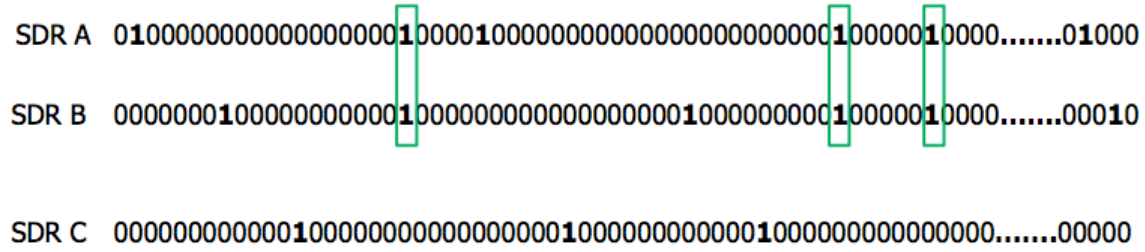


Figure 2: SDR A and SDR B have matching 1 bits and therefore have shared semantic meaning; SDR C has no matching bits or shared semantic meaning.

There are some surprising mathematical properties of SDRs that don't manifest in tradition computer data structures. For example, to store an SDR in memory, we don't have to store all the bits as you would with a dense representation. We only have to store the locations of the 1-bits, and surprisingly, we only have to store the locations of a small number of the 1-bits. Say we have an SDR with 10,000 bits, of which 2%, or 200, are 1's. To store this SDR, we remember the locations of the 200 1-bits. To compare a new SDR to the stored SDR, we look to see if there is a 1-bit in each of the 200 locations of the new SDR. If there is, then the new SDR matches the stored SDR. But now imagine we store the location of just 10 of the 1-bits randomly chosen from the original 200. To see if a new SDR matches the stored SDR, we look for 1-bits in the 10 locations. You might be thinking, "But wait, there are many patterns that would match the 10 bits yet be different in the other bits. We could easily have a false positive match!" This is true. However, the chance that a randomly chosen SDR would share the same 10 bits is extremely low; it won't happen by chance, so storing ten bits is sufficient. However, if two SDRs did have ten 1-bits in the same location but differed in the other bits then the two SDRs are semantically similar. Treating them as the same is a useful form of generalization. We discuss this interesting property, and derive the math behind it, later in this chapter.

Another surprising and useful property of SDRs is the union property, which is illustrated in Figure 3. We can take a set of SDRs and form a new SDR, which is the union of the original set. To form a union, we simply OR the SDRs together. The resulting union has the same number of bits as each of the original SDRs, but is less sparse. Forming a union is a one-way operation, which means that given a union SDR you can't say what SDRs were used to form the union. However, you can take a new SDR, and by comparing it to the union, determine if it is a member of the set of SDRs used to form the union. The chance of incorrectly determining membership in the union is very low due to the sparseness of the SDRs.

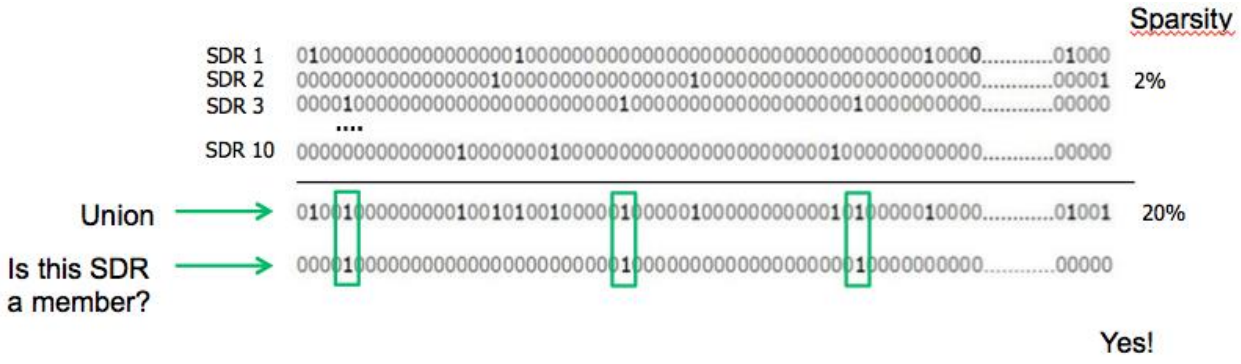


Figure 3: A union of 10 SDRs is formed by taking the mathematical OR of the bits. New SDR membership is checked by confirming 1 bits match. Note the union SDR is less sparse than the input SDRs.

These properties, and a few others, are incredibly useful in practice and get to the core of what makes brains different than computers. The following sections describe these properties and operations of SDRs in more detail. At the end of the chapter we discuss some of the ways SDRs are used in the brain and in HTMs.

Mathematical Properties of SDRs

In this section, we discuss the following mathematical properties of SDRs with a focus on deriving fundamental scaling laws and error bounds:

- Capacity of SDRs and the probability of mismatches
- Robustness of SDRs and the probability of error with noise
- Reliable classification of a list of SDR vectors
- Unions of SDRs
- Robustness of unions in the presence of noise

These properties and their associated operations demonstrate the usefulness of SDRs as a memory space, which we illustrate in examples relevant to HTMs. In our analysis we lean on the intuitions provided by Kanerva (Kanerva, 1988 & 1997) as well as some of the techniques used for analyzing Bloom filters (Bloom, 1970). We start each property discussion with a summary description, and then go into the derivation of the mathematics behind the property. But first, here are some definitions of terms and notations we use in the following discussion and throughout the text. A more comprehensive list of terms can be found in the Glossary at the end of this book.

Mathematical Definitions and Notation

Binary vectors: For the purposes of this discussion, we consider SDRs as **binary vectors**, using the notation $\mathbf{x} = [b_0, \ldots, b_{n-1}]$ for an SDR x. The values of each element are "0" or "1", for OFF and ON, respectively.

Vector size: In an SDR $\mathbf{x} = [b_0, \ldots, b_{n-1}]$, n denotes the **size of a vector**. Equivalently, we say n represents the total number of positions in the vector, the dimensionality of the vector, or the total number of bits.

Sparsity: At any point in time, a fraction of the n bits in vector **x** will be ON and the rest will be OFF. Let s denote the percent of ON bits. Generally in sparse representations, s will be substantially less than 50%.

Vector cardinality: Let w denote the **vector cardinality**, which we define as the total number of ON bits in the vector. If the percent of ON bits in vector **x** is s, then $w_{\mathbf{x}} = s \times n = \|\mathbf{x}\|_0$.

Overlap: We determine the similarity between two SDRs using an **overlap score**. The overlap score is simply the number of ON bits in common, or in the same locations, between the vectors. If **x** and **y** are two SDRs, then the overlap can be computed as the dot product:

$$overlap(\mathbf{x}, \mathbf{y}) \equiv \mathbf{x} \cdot \mathbf{y}$$

Notice we do not use a typical distance metric, such as Hamming or Euclidean, to quantify similarity. With overlap we can derive some useful properties discussed later, which would not hold with these distance metrics.

Matching: We determine a **match** between two SDRs by checking if the two encodings overlap sufficiently. For two SDRs **x** and **y**:

$$match(\mathbf{x}, \mathbf{y}|\theta) \equiv overlap(\mathbf{x}, \mathbf{y}) \geq \theta$$

If **x** and **y** have the same cardinality w, we can determine an exact match by setting $\theta = w$. In this case, if θ is less than w, the overlap score will indicate an **inexact match**.

Consider an example of two SDR vectors:

$$\mathbf{x} = [01000000000000000001000000000000110000000]$$

$$\mathbf{y} = [10000000000000000001000000000000110000000]$$

Both vectors have size $n = 40$, $s = 0.1$, and $w = 4$. The overlap between x and y is 3; i.e. there are three ON bits in common positions of both vectors. Thus the two vectors match when the threshold is set at $\theta = 3$, but they are not an exact match. Note that a threshold larger than either vector's cardinality – i.e., $\theta > w$ – implies a match is not possible.

Capacity of SDRs and the Probability of Mismatches

To be useful in practice, SDRs should have a large capacity. Given a vector with fixed size n and cardinality w, the number of unique SDR encodings this vector can represent is the combination n choose w:

$$\binom{n}{w} = \frac{n!}{w!\,(n-w)!} \tag{1}$$

Note this is significantly smaller than the number of encodings possible with dense representations in the same size vector, which is 2^n. This implies a potential loss of capacity, as the number of possible input patterns is much greater than the number of possible representations in the SDR encoding. Although SDRs have a much smaller capacity than dense encodings, in practice this is meaningless. With typical values such as $n = 2048$ and $w = 40$, the SDR representation space is astronomically large at 2.37×10^{84} encodings; the estimated number of atoms in the observable universe is $\sim 10^{80}$.

For SDRs to be useful in representing information we need to be able to reliably distinguish between encodings; i.e. SDRs should be distinct such that we don't confuse the encoded information. It is valuable then to understand the probability with which two random SDRs would be identical. Given two random SDRs with the same parameters, x and y, the probability they are identical is

$$P(x = y) = 1/\binom{n}{w} \tag{2}$$

Consider an example with $n = 1024$ and $w = 2$. There are 523,776 possible encodings and the probability two random encodings are identical is rather high, i.e. one in 523,776. This probability decreases extremely rapidly as w increases. With $w = 4$, the probability dives to less than one in 45 billion. For $n = 2048$ and $w = 40$, typical HTM values, the probability two random encodings are identical is essentially zero. Please note (2) reflects the false positive probability under exact matching conditions, not inexact matching used in most HTM models; this is discussed later in the chapter.

The equations above show that SDRs, with sufficiently large sizes and densities, have an impressive capacity for unique encodings, and there is almost no chance of different representations ending up with the same encoding.

Overlap Set

We introduce the notion of an overlap set to help analyze the effects of matching under varying conditions. Let **x** be an SDR encoding of size n with w_x bits on. The overlap set of **x** with respect to b is $\Omega_{\mathbf{x}}(n, w, b)$, defined as the set of vectors of size n with w bits on, that have exactly b bits of overlap with **x**. The number of such vectors is $|\Omega_{\mathbf{x}}(n, w, b)|$, where $|\cdot|$ denotes the number of elements in a set. Assuming $b \leq w_x$ and $b \leq w$,

$$|\Omega_x(n, w, b)| = \binom{w_x}{b} \times \binom{n - w_x}{w - b} \tag{3}$$

The first term in the product of (3) the number of subsets of x with b bits ON, and the second term is the number of other patterns containing $n - w_x$ bits, of which $w - b$ bits are ON.

The overlap set is instructive as to how we can compare SDRs reliably; i.e. not get false negatives or false positives, even in the presence of significant noise, where noise implies random fluctuations of ON/OFF bits. In the following sections we explore the robustness of SDRs in the presence of noise using two different concepts, inexact matching and subsampling.

Inexact Matching

If we require two SDRs to have an exact match (i.e. $\theta = w$) then even a single bit of noise in either of the SDRs' ON bits would generate a false negative where we fail to recognize matching SDRs. In general we would like the system to be tolerant to changes or noise in the input. That is, rarely would we require exact matches, where $\theta = w$. Lowering θ allows us to use inexact matching, decreasing the sensitivity and increasing the overall noise robustness of the system. For example, consider SDR vectors $\mathbf{x}$ and $\mathbf{x}'$, where $\mathbf{x}'$ is $\mathbf{x}$ corrupted by random noise. With $w = 40$ and θ lowered to 20, the noise can flip 50% of the bits (ON to OFF and vice-versa) and still match $\mathbf{x}$ to $\mathbf{x}'$.

Yet increasing the robustness comes with the cost of more false positives. That is, decreasing θ also increases the probability of a false match with another random vector. There is an inherent tradeoff in these parameters, as we would like the chance of a false match to be as low as possible while retaining robustness.

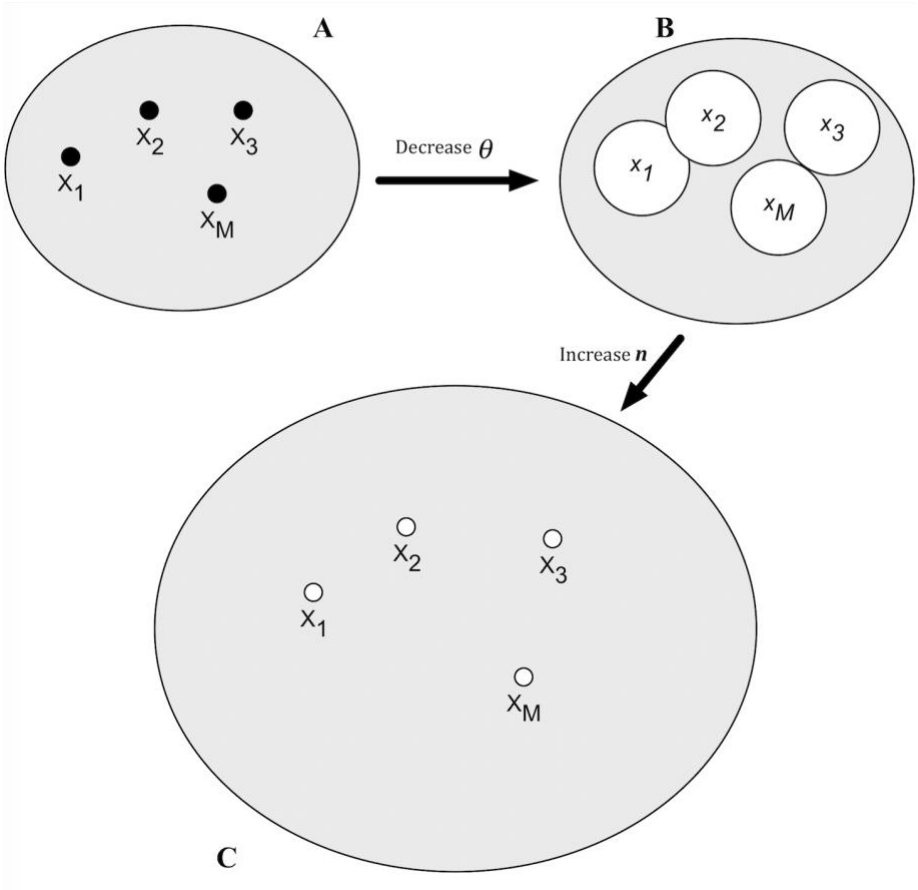


Figure 4: This figure illustrates the conceptual difference of lowering the match threshold θ. The large grey areas represent the space of all possible SDRs, where the elements $x_1, x_2, \ldots x_M$ are individual SDRs within the space. In space A we see the exact matching scenario, where θ=w and SDRs are single points in the space. As you decrease θ the set of potential matches increases. Notice $x_1, x_2, \ldots x_M$ in space B are now larger circles within the space, implying more SDRs will match to them in B than in A. Since the ratio of white to grey becomes much larger as you decrease θ, there is a greater chance of random false matches. Spaces A and B are the same size because they have a fixed n. If we increase n—i.e. increase the space of possible SDRs—the ratio of white to grey becomes smaller, as shown in space C. The transitions from A to B to C illustrate the tradeoff between the parameters θ and n: decreasing θ gives you more robustness, but also increases your susceptibility to false matches, but increasing n mitigates this effect.

With appropriate parameter values the SDRs can have a large amount of noise robustness with a very small chance of false positives. To arrive at the desired parameter values we need to calculate the false positive likelihood as a function of the matching threshold.

Given an SDR encoding **x** and another random SDR **y**, both with size n and cardinality w, what is the probability of a false match, i.e. the chance the $overlap(\mathbf{x}, \mathbf{y}) \geq \theta$? A match is defined as an overlap of θ bits or greater, up to w. With $\binom{n}{w}$ total patterns, the probability of a false positive is:

$$fp_w^n(\theta) = \frac{\sum_{b=\theta}^{w} |\Omega_x(n, w, b)|}{\binom{n}{w}} \tag{4}$$

What happens when $\theta = w$, or an exact match? The numerator in (4) evaluates to 1, and the equation reduces to (2).

To gain a better intuition for (4), again suppose vector parameters $n = 1024$ and $w = 4$. If the threshold is $\theta = 2$, corresponding to 50% noise, then the probability of an error is one in 14,587. That is, with 50% noise there is a significant chance of false matches. If w and θ are increased to 20 and 10, respectively, the probability of a false match decreases drastically to less than one in 10^{13}! Thus, with a relatively modest increase in w and θ, and holding n fixed, SDRs can achieve essentially perfect robustness with up to 50% noise. Figure 5 illustrates this for HTM values used in practice.

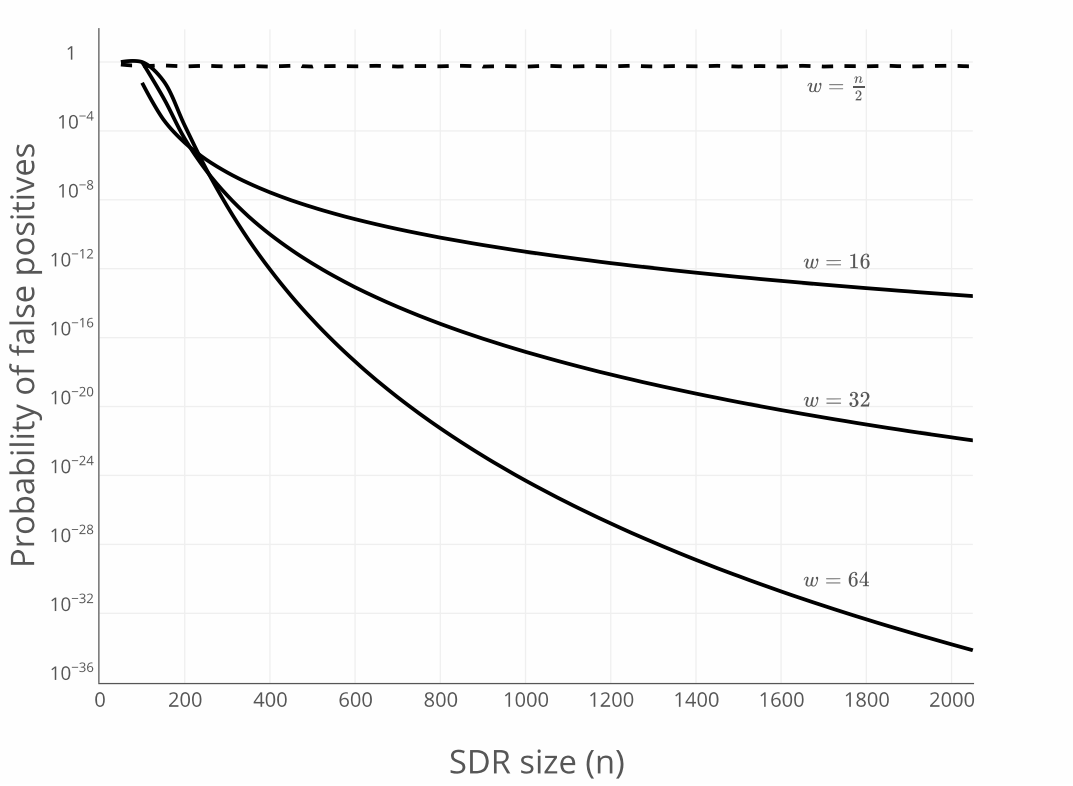


Figure 5: This plot illustrates the behavior of Eq. 4. The three solid curves show the rapid drop in error rates (i.e. probability of false positives) as you increase the SDR size n. Each curve represents a different number of ON bits w, and a constant 50% match threshold θ. For all three curves the error drops faster than exponentially as n increases, becoming essentially 0 once $n > 2000$. The dashed line shows the error rate when half of the bits in the SDR are ON. Notice this line maintains a relatively high error rate (around 50% error), implying it is not possible to get robust recognition with a non-sparse representation. Both sparsity and high-dimensionality are required to achieve low error rates.

Subsampling

An interesting property of SDRs is the ability to reliably compare against a subsampled version of a vector. That is, recognizing a large distributed pattern by matching a small subset of the active bits in the large pattern. Let **x** be an SDR and let $\mathbf{x}'$ be a subsampled version of **x**, such that $w_{x'} \leq w_x$. It's self-evident the subsampled vector $\boldsymbol{x}'$ will always match **x**, provided $\theta \leq w_{x'}$ that is. However, as you increase the subsampling the chance of a false positive increases.

What is the probability of a false match between $\mathbf{x}'$ and a random SDR **y**? Here the overlap set is computed with respect to the subsample $\mathbf{x}'$, rather than the full vector **x**. If $b \leq w_{x'}$ and $b \leq w_y$ then the number of patterns with exactly b bits of overlap with x' is:

$$\left|\Omega_{x'}(n, w_y, b)\right| = \binom{w_{x'}}{b} \times \binom{n - w_{x'}}{w_y - b} \tag{6}$$

Given a threshold $\theta \leq w_{x'}$, the chance of a false positive is then:

$$fp_{w_y}^{n}(\theta) = \frac{\sum_{b=\theta}^{w_{x'}} \left|\Omega_{x'}(n, w_y, b)\right|}{\binom{n}{w_y}} \tag{7}$$

Notice (6) and (7) differ from (3) and (4), respectively, only in the vectors being compared. That is, subsampling is simply a variant of the inexact matching properties discussed above.

For instance, suppose $n = 1024$ and $w_y = 8$. Subsampling half the bits in x and setting the threshold to two (i.e. $w_{x'} = 4, \theta = 2$), we find the probability of an error is one in 3,142. However, increasing w_y to 20 and the relevant parameter ratios fixed (i.e. $w_{x'} = 10, \theta = 5$) the chance of a false positive drops precipitously to one in 2.5 million. Increasing n to 2048, $w_y = 40$, $w_{x'} = 20$, and $\theta = 10$, more practical HTM parameter values, the probability of a false positive plummets to better than one in 10^{12}! This is remarkable considering that the threshold is about 25% of the original number of ON bits. Figure 6 illustrates this reliability in subsampling for varying HTM parameters.

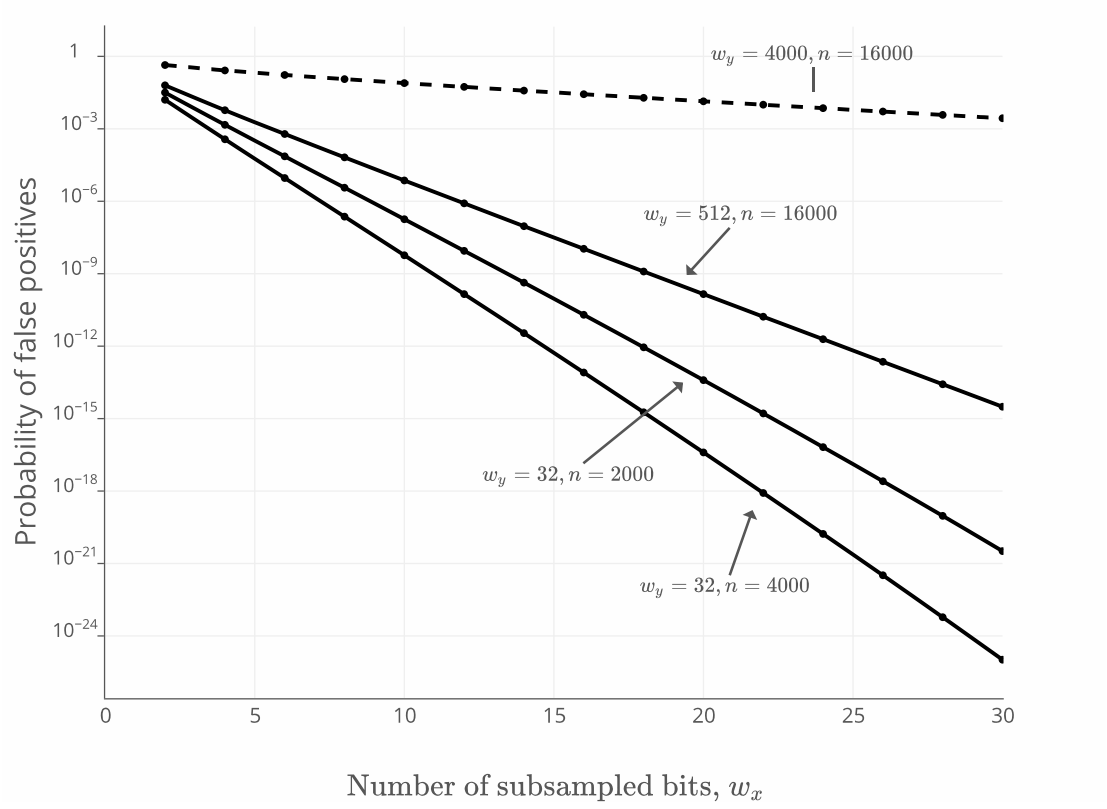


Figure 6: This plot illustrates the behavior of Eq. 6, where we can see the error rates (i.e. probability of false positives) as a function of the size (i.e. number of bits) of the subsampling. The three solid curves represent a few dimensionalities and sparsities, showing an exponential improvement in error rates as the number of subsampled bits increases. With sufficiently high dimensionality and sparse activity, subsampling values between 15 and 25 can lead to very low error rates. Conversely, the dashed line represents the error rates for a relatively dense representation (25% total ON bits); the error remains high, despite the high dimensionality.

With practical parameters, it is clear SDRs offer minimal probabilities of false positives. The properties of subsampling and inexact matching allow us to use SDRs as reliable mechanisms for classification, discussed in the next section.

Reliable Classification of a List of SDR Vectors

A useful operation with SDRs is the ability to classify vectors, where we can reliably tell if an SDR belongs to a set of similar SDRs. We consider a form of classification similar to nearest neighbor classification. Let X be a set of M vectors, $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_M\}$, where each vector $\boldsymbol{x}_i$ is an SDR. Given any random SDR $\boldsymbol{y}$ we classify it as belonging to this set as follows:

$$\exists_{\boldsymbol{x}_i \in X} match(\boldsymbol{x}, \boldsymbol{y}) = true \tag{8}$$

How reliably can we classify a vector corrupted by noise? More specifically, if we introduce noise in a vector $\boldsymbol{x}_i$ by toggling ON/OFF a random selection t of the n bits, what is the likelihood of a false positive classification? Assuming $t \leq w - \theta$, there are no false negatives in this scheme, only false positives. Thus the question of interest becomes what is the probability the classification of a random vector $\boldsymbol{y}$ is a false positive? Since all vectors in X are unique with respect to matching, the probability of a false positive is given by:

$$fp_x(\theta) = 1 - (1 - fp_w^n(\theta))^M \tag{9}$$

In practice the false positive probability of an individual overlap is extremely small and it is difficult to compute without numerical issues. We have found it is more practical to use the following bound:

$$fp_x(\theta) \leq Mfp_w^n(\theta) \tag{10}$$

Consider for example $n = 64$ and $w = 3$ for all vectors. If $\theta = 2$, 10 vectors can be stored in your list, and the probability of false positives is about one in 22. Increasing w to 12 and θ to eight, maintaining the ratio $\frac{\theta}{w} = \frac{2}{3}$, the chance of a false positive drops to about one in 2363. Now increase the parameters to more realistic values: $n = 1024$, $w = 21$, and $\theta = 14$ (i.e. two-thirds of w). In this case the chance of a false positive with 10 vectors plummets to about one in 10^{20}. In fact, with these parameters the false positive rate for storing a billion vectors is better than one in 10^{12}!

This result illustrates a remarkable property of SDRs. Suppose a large set of patterns is encoded in SDRs, and stored in a list. A massive number of these patterns can be retrieved almost perfectly, even in the presence of a large amount of noise. The main requirement being the SDR parameters n, w, and t need to be sufficiently high. As illustrated in the above example, with low values such as $n = 64$ and $w = 3$ your SDRs are unable to take advantage of these properties.

Unions of SDRs

One of the most fascinating properties of SDRs is the ability to reliably store a set of patterns in a single fixed representation by taking the OR of all the vectors. We call this the union property. To store a set of M vectors, the union mechanism is simply the Boolean OR of all the vectors, resulting in a new vector $\boldsymbol{X}$. To determine if a new SDR $\mathbf{y}$ is a member of the set, we simply compute the $match(\boldsymbol{X}, \mathbf{y})$.

$$\boldsymbol{x}_1 = [01000000000010000000 \ldots 010]$$

$$\boldsymbol{x}_2 = [00000000000000000010 \ldots 100]$$

$$\boldsymbol{x}_3 = [10100000000000000000 \ldots 010]$$

$$\vdots$$

$$\boldsymbol{x}_{10} = [00000000000000110000 \ldots 010]$$

$$\boldsymbol{X} = \boldsymbol{x}_1 OR \boldsymbol{x}_2 OR, \ldots, \boldsymbol{x}_{10}$$

$$\boldsymbol{X} = [11100000000110110000 \ldots 110]$$

Figure 7 (top) Taking the OR of a set of M SDR vectors results in the union vector **X**. With each individual vector having a total of 2% ON bits, and M = 10, it follows that the percentage of ON bits in **X** is at most 20%. The logic is straightforward: if there is no overlap within the set of vectors, each ON bit will correspond to its own ON bit in the union vector, summing the ON bit percentages. With overlap, however, ON bits will be shared in the union vector, resulting in a lower percentage of ON bits. *(bottom)* Computing the match(**X**, **y**) reveals if **y** is a member of the union set **X** – i.e. if the ON positions in **y** are ON in **X** as well.

The advantage of the union property is that a fixed-size SDR vector can store a dynamic set of elements. As such, a fixed set of cells and connections can operate on a dynamic list. It also provides an alternate way to do classification. In HTMs, unions of SDRs are used extensively to make temporal predictions, for temporal pooling, to represent invariances, and to create an effective hierarchy. However, there are limits on the number of vectors that can be reliably stored in a set. That is, the union property has the downside of increased potential for false positives.

How reliable is the union property? There is no risk of false negatives; if a given vector is in the set, its bits will all be ON regardless of the other patterns, and the overlap will be perfect. However, the union property increases the likelihood of false positives. With the number of vectors, M, sufficiently large, the union set will become saturated with ON bits, and almost any other random vector will return a false positive match. It's essential to understand this relationship so we can stay within the limits of the union property.

Let us first calculate the probability of a false positive assuming exact matches, i.e. $\theta = w$. In this case, a false positive with a new random pattern y occurs if all of the bits in y overlap with X. When $M = 1$, the probability any given bit is OFF is given by $1 - s$, where $s = \frac{w}{n}$. As M grows, this probability is given by:

$$p_0 = (1 - s)^M \tag{11}$$

After M union operations, the probability a given bit in X is ON is $1 - p_0$. The probability of a false positive, i.e. all w bits in y are ON, is therefore:

$$p_{fp} = (1 - p_0)^w = (1 - (1 - s)^M)^w \tag{12}$$

The technique used to arrive at (12) is similar to the derivation of the false positive rate for Bloom filters (Bloom, 1970; Broder and Mitzenmacher, 2004). The slight difference is that in Bloom filters each bit is chosen independently, i.e. with replacement. As such, a given vector could contain less than w ON bits. In this analysis we guarantee that there are exactly w bits ON in each vector.

The above derivation assures us, under certain conditions, we can store SDRs as unions without much worry for false positives. For instance, consider SDR parameters $n = 1024$ and $w = 2$. Storing $M = 20$ vectors, the chance of a false positive is about one in 680. However, if w is increased to 20, the chance drops dramatically to about one in 5.5 billion. This is a remarkable feature of the union property. In fact, if increasing M to 40, the chance of an error is still better than 10^{-5}.

To gain an intuitive sense of the union property, the expected number of ON bits in the union vector is $n(1 - p_0)$. This grows slower than linearly, as shown in (12); additional union operations contribute fewer and fewer ON bits to the resulting SDR. Consider for instance $M = 80$, where 20% of the bits are 0. When we consider an additional vector with 40 ON bits, there is a reasonable chance it will have at least one bit among this 20, and hence it won't be a false positive. That is, only vectors with all of their w bits amongst the 80% ON are false positives. As we increase n and w, the number of patterns that can OR together reliably increases substantially. As illustrated in Figure 8, if n and w are large enough, the probably of false positives remains acceptably low, even as M increases.

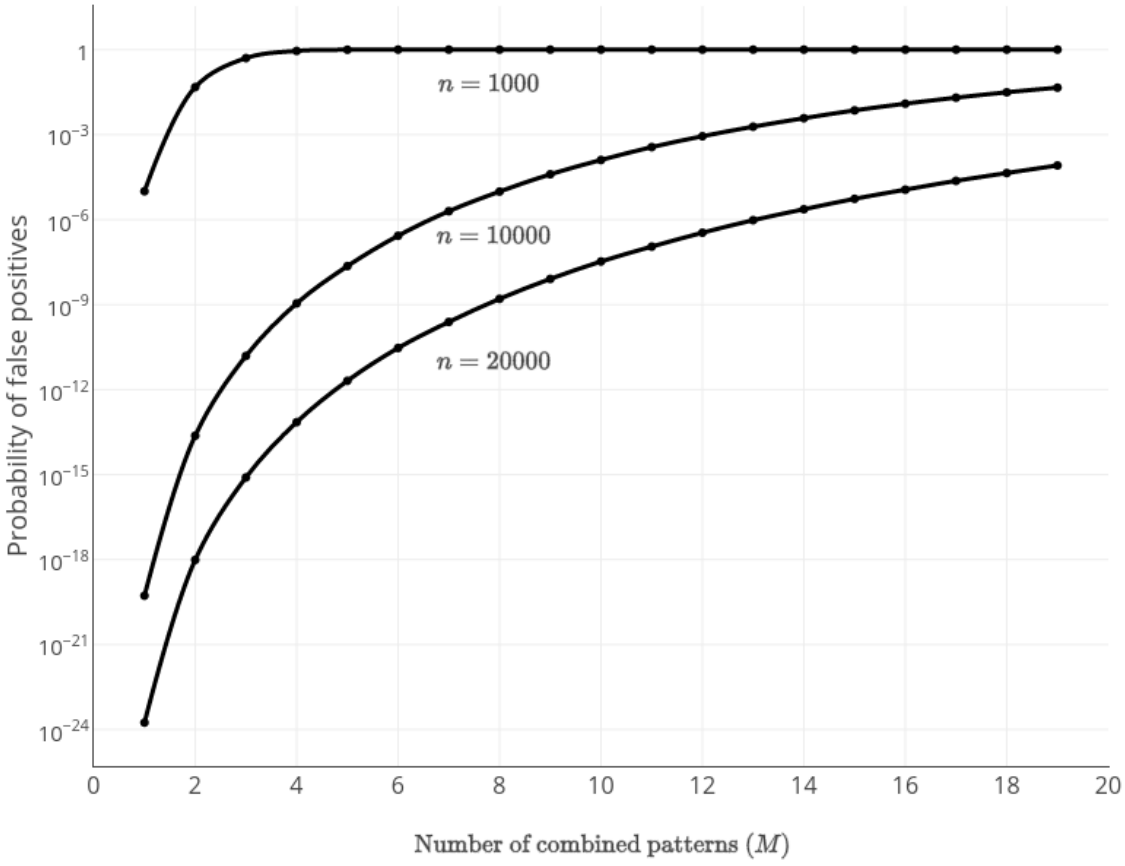


Figure 8: This plot shows the classification error rates (i.e. probability of false positives when matching to a union set of SDRs) of Eq. 12. The three lines show the calculated values for a few SDR dimensionalities, where $w = 200$. We see the error increases monotonically with the number of patterns stored. More importantly the plot shows that the size of the SDR is a critical factor: a small number of bits (1000) leads to relatively high error rates, while larger vectors (10,000+) are much more robust. Use this plot interactively online: *placeholder for plot.ly link*

Robustness of Unions in the Presence of Noise

As mentioned above, the expected number of ON bits in the union vector $\boldsymbol{X}$ is $\widetilde{w}_X = n(1 - p_0)$, where we use the tilde notation to represent a union vector. Assuming $n \geq \widetilde{w}_X \geq w$, we can calculate the expected size of the overlap set:

$$E[|\Omega_X(n, w, b)|] = \binom{\widetilde{w_X}}{b} \times \binom{n - \widetilde{w_X}}{w - b} \tag{13}$$

For a match we need an overlap of θ or greater bits (up to w). The probability of a false match is therefore:

$$\varepsilon \approx \frac{\sum_{b=\theta}^{w} |\Omega_X(n, w, b)|}{\binom{n}{w}} \tag{14}$$

Notice (14) is an approximation of the error, as we're working with the expected number of ON bits in $\boldsymbol{X}$[3].

As you would expect, the chance of error increases as the threshold is lowered, but the consequences of this tradeoff can be mitigated by increasing n. Suppose $n = 1024$ and $w = 20$. When storing $M = 20$ vectors, the chance of a false positive when using perfect matches is about one in 5 billion. Using a threshold of 19 increases the false positive rate to about one in 123 million. When $\theta = 18$, the chance increases to one in 4 million. However, if you increase n to 2048 with $\theta = 18$, the false positive rate drops dramatically to one in 223 billion! This example illustrates the union property's robustness to noise, and is yet another example of our larger theme: small linear changes in SDR numbers can cause super-exponential improvements in the error rates.

Computational Efficiency

Although SDR vectors are large, all the operations we've discussed run in time linear with the number of ON bits[4]. That is, the computations are dependent on the number of ON bits, w, not on the size of the vector, n. For HTM systems this is important since in practice $w \ll n$. In addition the vectors are binary, enabling fast calculations on most CPUs. This would not be the case, however, with more standard distance metrics, which are typically $O(n)$ and involve floating point calculations.

In the following section we discuss how the brain takes advantage of the mathematical properties of SDRs, and how this is manifested in HTM systems.

SDRs in the Brain and in HTM Systems

How do sparse distributed representations and their mathematical principles relate to information storage and retrieval in the brain? In this section we will list some of the ways SDRs are used in the brain and the corresponding components of HTM theory.

Neuron Activations are SDRs

If you look at any population of neurons in the neocortex their activity will be sparse, where a low percentage of neurons are highly active (spiking) and the remaining neurons are inactive or spiking very slowly. SDRs represent the activity of a set of neurons, with 1- and 0-bits representing active and relatively inactive neurons, respectively. All the functions of an HTM system are based on this basic correspondence between neurons and SDRs.

The activity of biological neurons is more complex than a simple 1 or 0. Neurons emit spikes, which are in some sense a binary output, but the frequency and patterns of spikes varies considerably for different types of neurons and under different conditions. There are differing views on how to interpret the output of a neuron. On one extreme are arguments that the timing of each individual spike matters; the inter-spike time encodes information. Other theorists consider the output of a neuron as a scalar value, corresponding to the rate of spiking. However, it has been shown that sometimes the neocortex can perform significant tasks so quickly that the neurons involved do not have enough time for even a second spike from each neuron to contribute to the completion of the task. In these tasks inter-spike timing and spike rate can't be responsible for encoding information. Sometimes neurons start

[3] The assumption behind this approximation being the actual number of ON bits is the same as the expected number of ON bits.
[4] The computations are O(w) and independent of the size of the vector, n.

spiking with a mini-burst of two to four spikes in quick succession before settling into a steadier rate of spiking. These mini-bursts can invoke long lasting effects in the post-synaptic cells, i.e. the cells receiving this input.

HTM theory says that a neuron can be in one of several states:

- Active (spiking)
- Inactive (not spiking or very slowly spiking)
- Predicted (depolarized but not spiking)
- Active after predicted (a mini-burst followed by spiking)

These HTM neuron states differ from those of other neural network models and this deserves some explanation. First, it is well established that some biological neurons spike at different rates depending on how well their input matches their ideal "receptive field". However, individual neurons are never essential to the performance of the network; the population of active cells is what matters most, and any individual neuron can stop working with little effect to the network. It follows that variable spiking rate is non-essential to neocortical function, and thus it is a property that we choose to ignore in the HTM model. We can always compensate for lack of variable encoding by using more bits in an SDR. All the HTM implementations created to date have worked well without variable rate encoding. The second reason for avoiding rate encoding is that binary cell states make software and hardware implementations much simpler. HTM systems require almost no floating point operations, which are needed in any system with rate encoding. Hardware for implementing HTM will be far simpler without the need for floating point math. There is an analogy to programmable computers. When people first started building programmable computers, some designers advocated for decimal logic. Binary logic won out because it is much simpler to build.

Although HTM neurons don't have variable rate outputs, they do incorporate two new states that don't exist in other theories. When a neuron recognizes a pattern on its distal or apical dendrites, the dendrite generates a local NMDA spike, which depolarizes the cell body without generating a somatic spike. In HTM theory, this internal depolarized state of the cell represents a prediction of future activity and plays a critical role in sequence memory. And finally, under some conditions a neuron will start firing with a mini-burst of spikes. One condition that can cause a mini-burst is when a cell starts firing from a previously depolarized state. A mini-burst activates metabotropic receptors in the post-synaptic cell, which leads to long lasting depolarization and learning effects. Although the neuroscience around mini-bursts is not settled, HTM theory has a need for the system acting differently when an input is predicted from when it isn't predicted. Mini-bursts and metabotropic receptors fill that role. By invoking metabolic effects the neuron can stay active after its input ceases and can exhibit enhanced learning effects. These two states, predicted (depolarized) and active after predicted (mini-burst) are excellent examples of how HTM theory combines top-down system-level theoretical needs with detailed biological detail to gain new insights into biology.

Figure 9: Visualization of an SDR in the brain *(placeholder)*

Neural Predictions as Unions of SDRs

HTM sequence memory makes multiple simultaneous predictions of what will happen next. This capability is an example of the union property of SDRs. The biological equivalent is multiple sets of cells (SDRs) being depolarized at the same time. Because each representation is sparse, many predictions can be made simultaneously. For example, consider a layer of neurons implementing an HTM sequence memory. If 1% of the neurons in the layer are active and lead to 20 different predictions, then about 20% of the neurons would be in the depolarized/predictive state. Even with 20% of neurons depolarized, the system can reliably detect if one of the predictions occurs or not. As a human, you would not be consciously aware of these predictions because the predicted cells are not spiking. However, if an unexpected input occurs, the network detects it, and you become aware something is wrong.

Synapses as a Means for Storing SDRs

Computer memory is often called "random access memory." A byte is stored in a memory location on a hard drive or on a memory chip. To access the byte's value you need to know its address in memory. The word "random" means that you can retrieve information in any order as long as you have the address of the item you want to retrieve. Memory in the brain is called "associative memory." In associative memory, one SDR is linked to another SDR which is linked to another, etc. SDRs are recalled through "association" with other SDRs. There is no centralized memory and no random access. Every neuron participates in both forming SDRs and in learning the associations.

Consider a neuron that we want to recognize a particular pattern of activity. To achieve this, the neuron forms synapses to the active cells in the pattern. As described above, a neuron only needs to form a small number of synapses, typically fewer than twenty, to

accurately recognize a pattern in a large population of cells as long as the pattern is sparse. Forming new synapses is the basis of almost all memory in the brain.

But we don't want just one neuron to recognize a pattern; we want a set of neurons to recognize a pattern. This way one SDR will invoke another SDR. We want SDR pattern "A" to invoke SDR pattern "B." This can be achieved if each active cell in pattern "B" forms twenty synapses to a random sample of the cells in pattern "A."

If the two patterns "A" and "B" are subsequent patterns in the same population of neurons then the learned association from "A" to "B" is a transition, and forms the basis of sequence memory. If the patterns "A" and "B" are in different populations of cells then pattern "A" will concurrently activate pattern "B." If the neurons in pattern "A" connect to the distal synapses of neurons in pattern "B," then the "B" pattern will be predicted. If the neurons in pattern "A" connect to the proximal synapses of neurons in pattern "B," then the "B" pattern will consist of active neurons.

All associative memory operations use the same basic memory mechanism, the formation of new synapses on the dendrite segments of neurons. Because all neurons have dozens of dendrite segments and thousands of synapses, each neuron does not just recognize one pattern but dozens of independent patterns. Each neuron participates in many different SDRs.

Conclusion

SDRs are the language of the brain, and HTM theory defines how to create, store, and recall SDRs and sequences of SDRs. In this chapter we learned about powerful mathematical properties of SDRs, and how these properties enable the brain, and HTM, to learn sequences and to generalize.

References

Kanerva, P. (1988). Sparse Distributed Memory. Bradford Books of MIT Press.

Kanerva, P. (1997). Fully distributed representation. Proceedings of 1997 Real World Computing Symposium (RWC '97, Tokyo, Jan. 1997), pp. 358–365. Tsukuba-city, Japan: Real World Computing Partnership.

Bloom, B.H. (1970). Space/ Time Trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, Volume 13, Number 7, July 1970, pp. 422-426.

Broder, A., & Mitzenmacher, M. (2004). Network Applications of Bloom Filters, A Survey. Internet Mathematics, Volume 1, No. 4, pp. 485-509.

Ahmad, S., & Hawkins, J. (2016). How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites. *arXiv*, 1601.00720. Neurons and Cognition; Artificial Intelligence. Retrieved from http://arxiv.org/abs/1601.00720

Encoders

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	S. Purdy

Encoding Data for HTM Systems

In this chapter we describe how to encode data as Sparse Distributed Representations (SDRs) for use in HTM systems. We explain several existing encoders, which are available through the open source project called NuPIC[5], and we discuss requirements for creating encoders for new types of data.

What is an Encoder?

Hierarchical Temporal Memory (HTM) provides a flexible and biologically accurate framework for solving prediction, classification, and anomaly detection problems for a broad range of data types (Hawkins and Ahmad, 2015). HTM systems require data input in the form of Sparse Distributed Representations (SDRs) (Ahmad and Hawkins, 2016). SDRs are quite different from standard computer representations, such as ASCII for text, in that meaning is encoded directly into the representation. An SDR consists of a large array of bits of which most are zeros and a few are ones. Each bit carries some semantic meaning, so if two SDRs have more than a few overlapping one-bits, then those two SDRs have similar meanings.

Any data that can be converted into an SDR can be used in a wide range of applications using HTM systems. Consequently, the first step of using an HTM system is to convert a data source into an SDR using what we call an encoder. The encoder converts the native format of the data into an SDR that can be fed into an HTM system. The encoder is responsible for determining which output bits should be ones, and which should be zeros, for a given input value in such a way as to capture the important semantic characteristics of the data. Similar input values should produce highly overlapping SDRs.

The Encoding Process

The encoding process is analogous to the functions of sensory organs of humans and other animals. The cochlea, for instance, is a specialized structure that converts the frequencies and amplitudes of sounds in the environment into a sparse set of active neurons (Webster et al, 1992; Schuknecht, 1974). The basic mechanism for this process (Fig. 1) comprises a set of inner hair cells organized in a row that are sensitive to different frequencies. When an appropriate frequency of sound occurs, the hair cells stimulate neurons that send the signal into the brain. The set of neurons that are triggered in this manner comprise the encoding of the sound as a Sparse Distributed Representation.

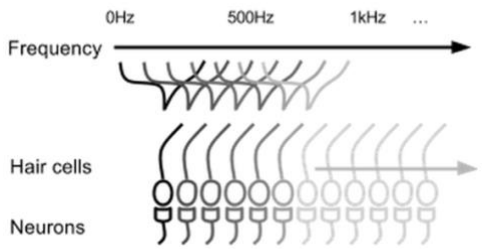


Figure 1 Cochlear hair cells stimulate a set of neurons based on the frequency of the sound.

[5] http://numenta.org

One important aspect of the cochlear encoding process is that each hair cell responds to a range of frequencies, and the ranges overlap with other nearby hair cells. This characteristic provides redundancy in case some hair cells are damaged but also means that a given frequency will stimulate multiple cells. And two sounds with similar frequencies will have some overlap in the cells that are stimulated. This overlap between representations is how the semantic similarity of the data is captured in the representation. It also means that the semantic meaning is distributed across a set of active cells, making the representation tolerant to noise or subsampling.

The cochleae for different animals respond to different ranges of frequencies and have different resolutions for which they can distinguish differences in the frequencies. While very high frequency sounds might be important for some animals to hear precisely, they might not be useful to others. Similarly, the design of an encoder is dependent on the type of data. The encoder must capture the semantic characteristics of the data that are important for your application. Many of the encoder implementations in NuPIC take range or resolution parameters that allow them to work for a broad range of applications.

There are a few important aspects that need to be considered when encoding data:

1. Semantically similar data should result in SDRs with overlapping active bits.
2. The same input should always produce the same SDR as output.
3. The output should have the same dimensionality (total number of bits) for all inputs.
4. The output should have similar sparsity for all inputs and have enough one-bits to handle noise and subsampling.

In following sections we will examine each of these characteristics in detail and then describe how you can encode several different types of data. Note that several SDR encoders exist already and most people will not need to create their own. Those who do should carefully consider the above criteria.

1) Semantically similar data should result in SDRs with overlapping active bits

To create an effective encoder, you must understand the aspects of your data that should contribute to similarity. In the cochlea example above, the encoder was designed to make sounds with similar pitch have similar representations but did not take into account how loud the sounds were, which would require a different approach.

The first step to designing an encoder is to determine each of the aspects of the data that you want to capture. For sound, the key features may be pitch and amplitude; for dates, it may be whether or not it is a weekend.

The encoder should create representations that overlap for inputs that are similar in one or more of the characteristics of the data that were chosen. So for weekend encoders, dates that fall on Saturdays and Sundays should overlap with each other, but not as much or at all with dates that fall on weekdays.

Preserving Semantics: A Formal Description

Here we formalize the encoding process by defining a set of rules that relate the semantic similarity of two inputs with the number of overlapping one-bits in the corresponding encoded SDRs.

Let $\mathcal{A}$ be an arbitrary input space and let $S(n,k)$ be the set of SDRs of length n with k ON bits. An encoder f is simply a function $f : \mathcal{A} \to S(n,k)$. A distance score $d_{\mathcal{A}}$ over space $\mathcal{A}$ is a function $d_{\mathcal{A}} : \mathcal{A} \times \mathcal{A} \to \mathbb{R}$ that satisfies three conditions:

1. $\forall x, y \in \mathcal{A}, d_{\mathcal{A}}(x,y) \geq 0$
2. $\forall x, y \in \mathcal{A}, d_{\mathcal{A}}(x,y) = d_{\mathcal{A}}(y,x)$
3. $\forall x \in \mathcal{A}, d_{\mathcal{A}}(x,x) = 0$

Equation 1 requires the semantic similarity metric give a distance value of zero or greater. Equation 2 requires the distance metric to be symmetric. And Equation 3 requires that the distance between two identical values be zero.

Given an input space and a distance score, we can evaluate an encoder by comparing the distance scores of pairs of inputs with the overlaps of their encodings. Two inputs that have low distance scores should have SDRs with high overlap, and vice versa. Moreover, if two SDRs have higher overlap than two other SDRs, then the former's pre-encoding distance score should be lower than the latter's. We state this formally below.

For SDRs s and t with the same length, let $O(s,t)$ be the number of overlapping bits (i.e. the number of ON bits in $s\&t$). Then for an encoder $f : \mathcal{A} \to S(n,k)$ and $\forall w, x, y, z \in \mathcal{A}$,

4. $O(f(w), f(x)) \geq O(f(y), f(z)) \Leftrightarrow d_{\mathcal{A}}(w,x) \leq d_{\mathcal{A}}(y,z)$

Equation 4 states that encodings with more overlapping one bits means the values have greater semantic similarity and, inversely, that values with greater semantic similarity will have encodings with more overlapping one bits. It is not always possible to create an encoder that satisfies this, but the equation can be used as a heuristic to evaluate the quality of an encoder.

2) The same input should always produce the same SDR as output

Encoders should be deterministic so that the same input produces the same output every time. Without this property, the sequences learned in an HTM system will become obsolete as the encoded representations for values change. Avoid creating encoders with random or adaptive elements.

It can be tempting to create adaptive encoders that adjust representations to handle input data with an unknown range. There is a way to design an encoder to handle this case without changing the representations of inputs that is described below in the section labeled "A more flexible encoder method." This method allows encoders to handle input with unbounded or unknown ranges.

3) The output should have the same dimensionality (total number of bits) for all inputs

The output of an encoder must always produce the same number of bits for each of its inputs. SDRs are compared and operated on using a bit-by-bit assumption such that a bit with a certain "meaning" is always in the same position. If the encoders produced varying bit lengths for the SDRs, comparisons and other operations would not be possible.

4) The output should have similar sparsity for all inputs and have enough one-bits to handle noise and subsampling

The fraction of total ON bits in an encoder can vary from around 1% to 35%, but the sparsity should be relatively fixed for a given application of an encoder. While keeping the sparsity the same should be the rule, small variations in sparsity will not have a negative effect.

Additionally, there must be enough one-bits to handle noise and subsampling. A general rule of thumb is to have at least 20-25 one bits. Encoders that produce representations with fewer than 20 one bits do not work well in HTM systems since they may become extremely susceptible to errors due to small amounts of noise or non-determinism.

Example 1 – Encoding Numbers

One of the most common data types to encode is numbers. This could be a numeric value of any kind – 82 degrees, $145 in sales, 34% of capacity, etc. The sections below describe increasingly-advanced encoders for a single numeric value. In each section, we will change the semantic characteristics that we desire to capture and update our encoding to achieve the new goal.

A Simple Encoder for Numbers

In the simplest approach, we can mimic how the cochlea encodes frequency. The cochlea has hair cells that respond to different but overlapping frequency ranges. A specific frequency will stimulate some set of these cells. We can mimic this process by encoding overlapping ranges of real values as active bits (Fig. 2). So the first bit may be active for the values 0 to 5, the next for 0.5 to 5.5, and so on. If we choose for our encoding to have 100 total bits, then the last bit will represent 49.5 to 54.5. The entire range of values that are represented will be 0 to 54.5. The minimum and maximum values in this approach are fixed. Values above or below the allowed range will be given a representation matching the maximum or minimum possible representations.

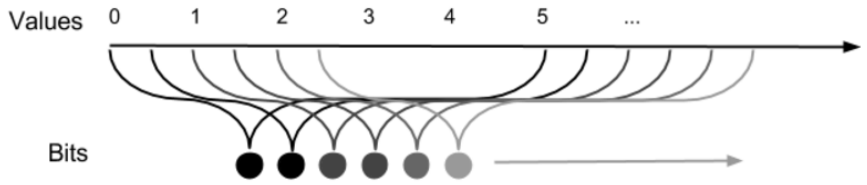


Figure 2 Each bit in the representation responds to a range of values that overlaps with its neighbors.

Using the encoder parameters from before, here is what the encoding of the values 7.0 (Fig. 3A), 10.0 (Fig. 3B), and 13.0 (Fig. 3C) would look like. Note that numbers that are close together (7.0 and 10.0, or 10.0 and 13.0) share one bits with each other, but numbers that are not so close together (7.0 and 13.0) do not share any one bits.

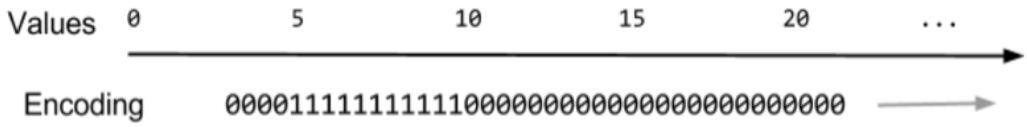


Figure 1A The encoded representation of 7.0 using encoding parameters of 100 total bits, minimum-value 0, value range per bit of 5.0, and increase per bit of 0.5.

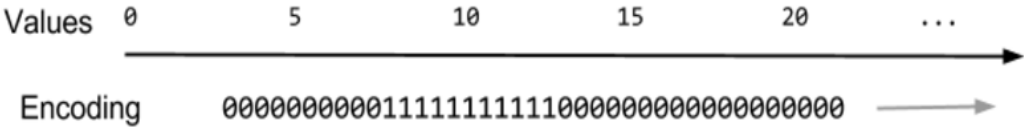


Figure 3B The encoded representation of 10.0 using encoding parameters of 100 total bits, minimum-value 0, value range per bit of 5.0, and increase per bit of 0.5. There are several shared one bits with the representation for 7.0.

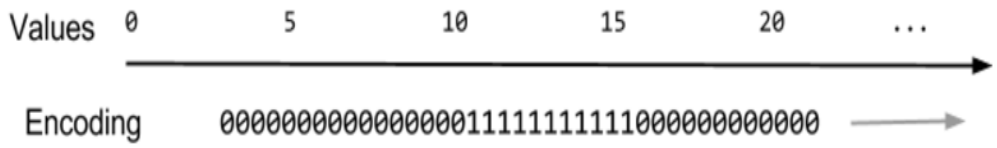


Figure 3C The encoded representation of 13.0 using encoding parameters of 100 total bits, minimum-value 0, value range per bit of 5.0, and increase per bit of 0.5. There are several shared one bits with the representation for 10.0, but none with the representation for 7.0.

When we create an implementation of this encoder, we first split the range of values into buckets, and then map the buckets to a set of active cells. Here are the steps for encoding a value with this approach:

1. Choose the range of values that you want to be able to represent, minVal and maxVal.
2. Compute the range as $range = maxVal - minVal$
3. Choose a number of buckets into which you will split the values.
4. Choose the number of active bits to have in each representation, w.
5. Compute the total number of bits, n: $n = buckets + w - 1$
6. For a given value, v, determine the bucket, i, that it falls into: $i = floor[buckets * (v - minVal)/range]$
7. Create the encoded representation by starting with n unset bits and then set the w consecutive bits starting at index i to active.

Note that this encoding scheme has four parameters: minimum value, maximum value, number of buckets, and number of active bits (w). Alternatively, you may choose the total number of bits, n, rather than the number of buckets, which would then be computed as $buckets = n - w + 1$.

Here is an example of encoding the outside temperature for a location where the temperature varies between 0°F and 100°F.

1. This minVal is 0°F and the maxVal is 100°F.
2. The range is 100.
3. We choose to split the range into 100 buckets.
4. We choose to have 21 active bits for each representation.
5. The total number of bits is computed to be $n = buckets + w - 1 = 120$
6. Now we can select the bucket for the value 72°F as $i = floor[100 * (72 - 0)/100] = 72$
7. And the representation will be 120 bits with 21 consecutive active bits starting at the 72nd bit:

000000000 … 000011111111111111111111100000000 … 00000000
72nd bit ↗

This encoding approach is simple and provides quite a bit of flexibility but requires that you know the appropriate range of the data. If your data falls outside the minimum and maximum values then the encoder doesn't work well. Typically you would use the smallest bucket for values below the range and the largest bucket for values above the range. Thus all values above the range will have the same representation, and similarly for those below the range. Below is the representation for a temperature of 100°F. This will also be the representation for 110°F or any other number larger than the maximum value of 100°F.

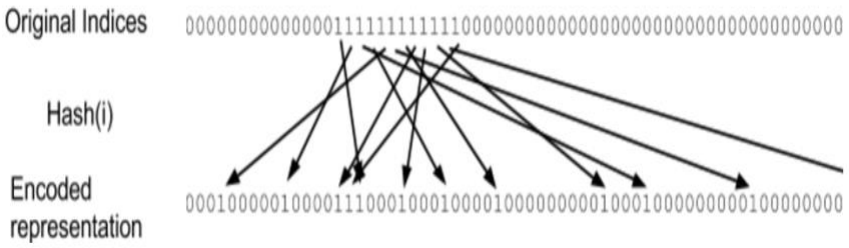
000000000000000 … 000000000000000000011111111111111111
100th bit ↗

The number of buckets can be chosen depending on the inherent noise for this metric in your application and the quality of the desired predictions. For a very noisy signal you might want a smaller number of buckets. This would allow the HTM to see more stable inputs, but it would not be able to make very precise predictions. For a very clean signal you could have a large number of buckets. In this case the HTM would be able to make very precise predictions.

A More Flexible Encoder for Numbers

Biological sensory organs such as the cochlea have a fixed range of values they can encode. In humans this is approximately 20 hertz to 20 kilohertz. If sounds in the world shifted to higher frequencies then our ears would be useless. The encoder mechanism we described above also has a fixed range. The designer of the encoder selects the range, but once it is chosen and the system starts learning then you can't change it. However, there is a way to overcome this limit. It is possible to design encoders that have a fixed number of bits but can encode an essentially limitless range of values by using a hash function[6]. With this design, each bit in an SDR can represent multiple ranges of values. If these ranges are assigned via a hash function then the SDRs for two values that are far apart may overlap by one or two bits, but this small overlap will not cause a problem for the HTM. We will now show how this works in a numeric encoder, but the same principle is used in the geospatial encoder described later in the chapter.

If we look at step 6 in the previous section, we see that each bucket is identified by a specific bit. We then select bits for the following w-1 buckets to complete the representation. Each bucket has overlapping bits with its neighbors. For a more flexible numeric encoder, we can do the same steps 1-6 but change how we select the bits in the representation. Specifically, we can use a hashing function to deterministically select one of the output bits from the bucket index. We do this selection separately for each bit, so rather than w consecutive bits being active, there are w bits active based on the hashing function. The hashing operation looks like this:

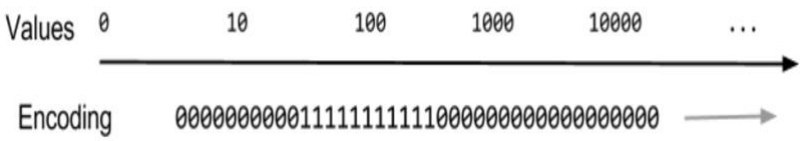


The advantage of this method is that you don't need to restrict the values to an overall range. Instead, you just need to select the range for which each bucket is responsible. It is possible that there will be some unwanted collisions. In other words, two buckets that should not have overlapping bits may actually have one or two overlapping bits due to chance. This issue is not a problem in practice as long as you choose your hash function carefully and have large enough w and n values.

If you take a single bit from this representation, you cannot know what the original value is since there are many different values that will produce encodings that include that bit. But if you take all of the active bits together, you have a representation very specific to your input. This may differ from many biological encoders, but it will work well because it satisfies the encoding properties that we outlined previously.

The Numeric Log Encoder

Some applications may benefit from numeric encoders that capture similarity between numbers differently based on how large the number is. In other words, the values 4000 and 5000 may be treated just as similar as the numbers 4 and 5. An encoding approach using a logarithmic function would look something like this:



This encoder is sensitive to small changes for small numbers. A change from 3 to 15 will result in a fairly different encoding. But the encoding for larger numbers is much less sensitive. A change from 1000 to 2000, for instance, would have a much smaller impact on the encoding despite being a much larger absolute change in the number.

[6] https://en.wikipedia.org/wiki/Hash_function

The Delta Encoder

A delta encoder is designed to capture the semantics of the change in a value rather than the value itself. This technique is useful for modeling data that has patterns that can occur in different value ranges and may be helpful to use in conjunction with a regular numeric encoder. A simple example of where you might use a delta encoder is when trying to predict a value that is constantly increasing. The predictable pattern in data with this characteristic is not the values themselves, but the change in the values – is the value larger or smaller than the previous value, and by how much? While a regular numeric encoder would give different representations for each new value, the delta encoder would produce overlapping encodings for values that increased or decreased from the previous value by a similar amount.

If you had a temperature sensor on a piece of machinery, the delta encoder would recognize temperature patterns caused by machine behavior even if the absolute range was different due to the surrounding weather patterns. The outside temperature may be 60 degrees one day and 70 the next. A regular numeric encoder's representations for values in the 60s would not share much, if any, overlap with the representations for values in the 70s. But the delta encoder would produce similar encodings when the values change in a similar way, even if the values themselves have never been seen before.

This encoder is different from the previous encoders since it uses both the current and the previous inputs to determine the output. The implementation of this encoder is the same as one of the other scalar encoders but you apply the encoder to the difference between the current input value and the previous.

Example 2 – Encoding Categories

Many datasets contain categorical information. In some cases the data consists of discrete, completely unrelated categories (such as the SKUs of products in a store). In other cases, the data consists of categories that may have some relation (such as days of a week). The examples below show how to encode these types of data.

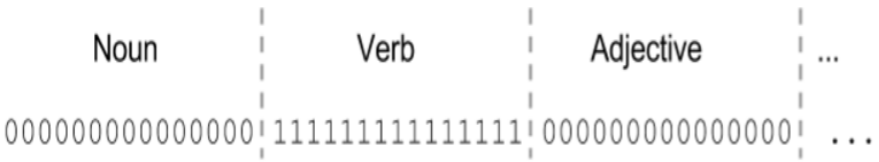
Some characteristics of dates and times are categorical in nature. For instance:

- Weekday vs weekend
- Holiday vs non-holiday
- Day vs night
- Meal time vs not meal time
- Part of speech for a word

In many cases it is useful to encode these characteristics as completely discrete categories. The encoding in these cases should attempt to minimize overlap between any of the category encodings. The easiest way to do this is dedicate some number of bits to each option. The encoding for any option has its dedicated bits active and the rest inactive. Here is an example of the weekday/weekend encoding for Saturday:

This encoding is useful to add to data streams in which the patterns in the stream are different during the week than on weekends. Adding this encoding ensures that the HTM systems receive distinct input patterns on weekdays vs. weekends, allowing them to more easily learn separate predictions for the two periods.

The category encoding can also be applied to part of speech. Here is what one such encoding may look like for a verb:



Ordered and Cyclic Categories

Sometimes the patterns change continuously rather than having completely discrete categories.

The difference, then, may not have a hard, discrete line. The patterns at noon may sometimes resemble the morning patterns, but other days may be more similar to the common evening patterns. In these cases, you simply convert to the numeric value and use a scalar encoder to generate the SDR.

However, particularly in the case of dates and times, the categories may be cyclic in addition to continuous. For example, Friday is a weekday, but a bit different than Thursday. Sunday evening is a weekend, but not quite like Saturday evening. You can start by representing the day of the week as an integer from 0 to 6, where 0 is Sunday and 6 is Saturday. A numeric encoder would create a good representation, except that there would be little or no overlap in the encodings for Saturday and Sunday since they are on opposite ends of the range. In these cyclic cases, the encoding must "wrap." For this example, we will use a small number of bits in the encoding. In a real implementation you would want more bits active and would need more total bits as a result. The easiest way to understand this is with diagrams (Fig. 4):

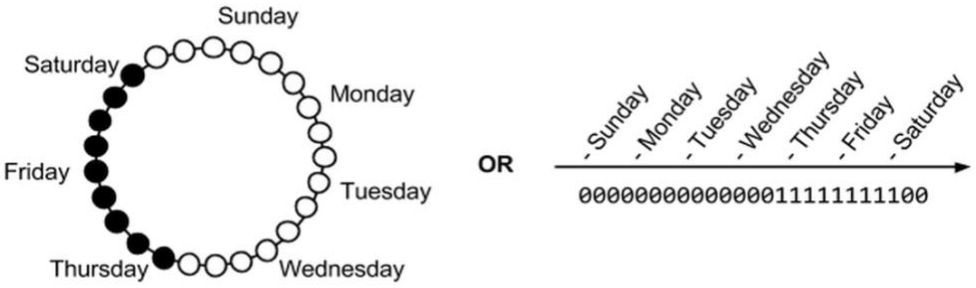


Figure 4 The encoding for a date that falls on a Friday.

Both diagrams in Fig. 4 show a representation of the encoding for Friday. The left diagram shows how each day falls on a circle and overlaps with days before and after it. The right diagram shows the encoded representation of Friday along with annotations for where the center of each encoding falls. The encoding for Sunday would include bits both at the very beginning and at the very end (Fig. 5).

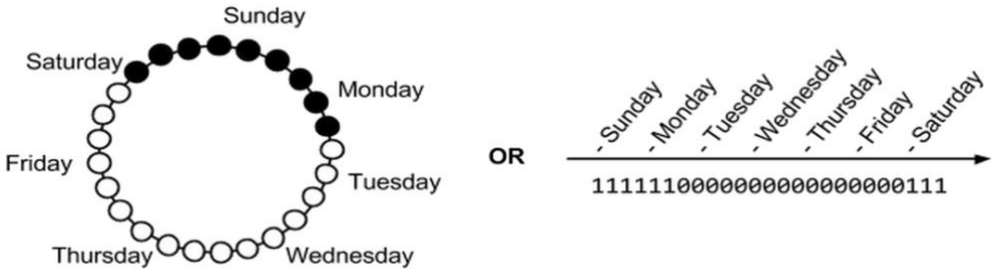


Figure 5 The encoding for a date that falls on a Sunday.

Other numeric characteristics of dates and times that you may want to capture are:

- Month of the year
- Day of the month
- Time of day
- Minute of the hour

Example 3 – Encoding Geospatial Data

A Simple Encoder for Geospatial Data

This example shows how an encoder can capture geospatial data. The most obvious aspect is that locations close to each other should be considered similar while locations far apart should not be considered similar. To encode this meaning, we first have to determine the resolution that we want to encode. For this example, we will assume that we are using GPS coordinates that are accurate to about ten feet. And we will be doing two dimensional locations, although extending the encoding to three dimensions would be straight-forward.

The first step is to convert the GPS locations into a flat space that we can block off into ten-by-ten foot squares. Then we decide on an indexing system so that we can identify any section by integer X and Y coordinates[7].

Now we need to encode a location as a set of n bits with w active. For this example we will use n=100 and w=25, but real applications would want to use larger numbers, like n=1000. We first select the coordinates for the location we want to encode, say x=5 and y=10, and then identify the coordinates for the surrounding locations. These will form a square set of locations identified by having the coordinates 3<=x<=7 and 8<=y<=12. This process gives us 25 sets of x, y coordinates shown in gray here:

...	8	9	10	11	12	13	14	15	...
2									
3									
4									
5									
6									
7									
8									
9									
...									

Now we can use a deterministic hash function to map each pair of coordinates to the 100 bits in the encoding:

$$\text{Hash}(x, y) = i_{x,y}$$

As explained earlier, by using a hash function we can use a fixed number of bits to represent any location in an unbounded space.

Because we are using a deterministic hashing function, we can compute these values when needed and do not need to store them. The encoding for one particular GPS coordinate ends up looking like this:

00000010010000000001010000000000000000110001000000000000…

Note that the final encoding may have slightly fewer than 25 one bits due to hash collisions. When using sufficiently large numbers for n and w it is unlikely that this will happen and in any case it will not create problems in the HTM.

If we encode the position at x=6, y=10, most (20 out of 25) of the selected coordinates will overlap with the encoding for x=5, y=10, yielding the result we want of semantic overlap in SDRs.

A More Flexible Encoder for Geospatial Data

The previously described encoding method works well if you want as fine an encoding as the geospatial system allows. If this is 10 feet, then moving just 20 feet away will result in a slightly different SDR and moving a thousand feet will result in a completely different SDR. But this might not be want you want. You may want an encoder that encodes small differences when an object is moving slowly and larger distances when moving fast. For example if someone is walking, then we may want the encoder to represent changes in position as small as ten feet, but if someone is in a car moving at highway speeds, then ten feet differences in the direction of travel is not important. At high speed maybe positions within 200 feet are best represented by the same SDR and positions separated by 1,000 feet should be overlapping. This type of variable encoding is desired in many geospatial applications.

[7] The spherical mercator is one projection that can be used to transform GPS coordinates into two dimensional coordinates: https://en.wikipedia.org/wiki/Mercator_projection

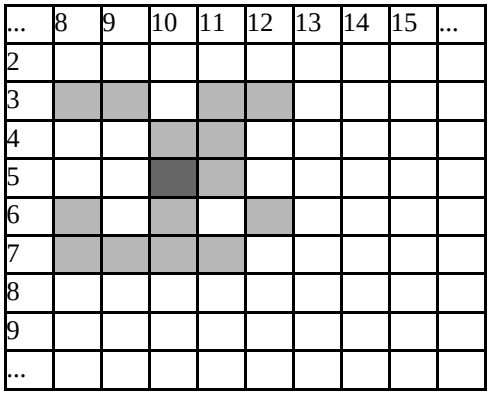
Usually there is a way to design an encoder for any need. In this case, the requirements can be met by forming an SDR using a subsample of all possible location bits and using the speed of the object to determine the coarseness of the sub-sampling.

The previously described geospatial encoding requires that the distance of two positions that you want to have overlap determine the number of active bits. You may not always want that many bits active. We can subsample from the range but need to determine which bits to subsample in a way that preserves the desired encoder properties. Specifically, we want data points that are close in space to have encodings that share bits. If we randomly subsample 50% of the bits inside the radius, it is possible that the encoding for a nearby position shares no bits even though their radii overlap.

One way to solve this problem is to give a strict ordering to all coordinates that determines which bits to subsample. To do this, we can map each coordinate to a floating point number between 0.0 and 1.0 using a deterministic hashing function:

$$\text{Hash}(x, y) = w_{x,y}$$

Because we are using a deterministic hashing function, these weights can be computed when needed and do not need to be stored in memory. Here is what our previous encoding might look like if we wanted fifteen active bits in the encoding:



The weighting scheme helps preference the same coordinates so that the encodings for nearby positions select some of the same coordinates but it doesn't guarantee this. As a result, picking a sufficiently high number of total and active bits is necessary to ensure proper overlap.

Incorporating Speed in Geospatial Encoders

One promising application of coordinate or geospatial encoders is anomaly detection on people or vehicle positions. The speed of a person or vehicle is important to know when choosing how far apart positions should be before they no longer share overlap. One way to automate this process is to dynamically adjust the radius to select bits based on the current speed, changing the distance semantics to be relative to your speed rather than being an absolute metric.

The following figures (Fig. 6A, 6B and 6C) show representations of the encodings for three positions. The first two encodings are for positions for the entity while it is moving slowly, while the third represents a third position after the entity has sped up. Note that while the third position is much further away, it still shares a similar number of bits with the previous encoding.

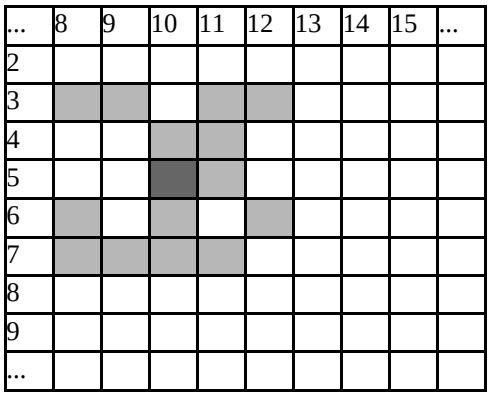


Figure 6A This is an encoding for a position in which the entity is moving slowly. The dark gray square represents the current location, X. The light gray squares are the ones chosen to be included in the representation from a radius of 2. These 15 squares have the highest weights of the 25 within the radius.

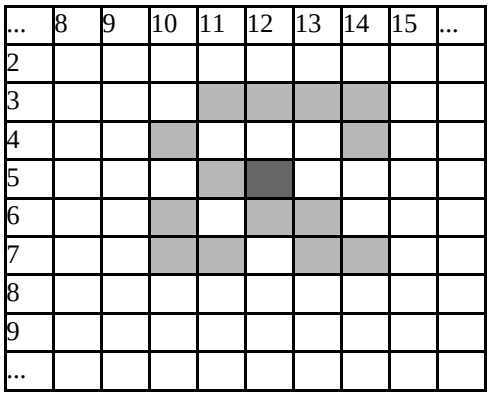


Figure 6B The encoding for a position nearby to X while the entity is still moving at the same speed. The entity has moved two coordinates to the right. For the coordinates that are within the radius for both positions, most of the same squares are selected because the weight for each square is fixed across encodings. And the radius is the same as X because the entity is moving at the same speed.

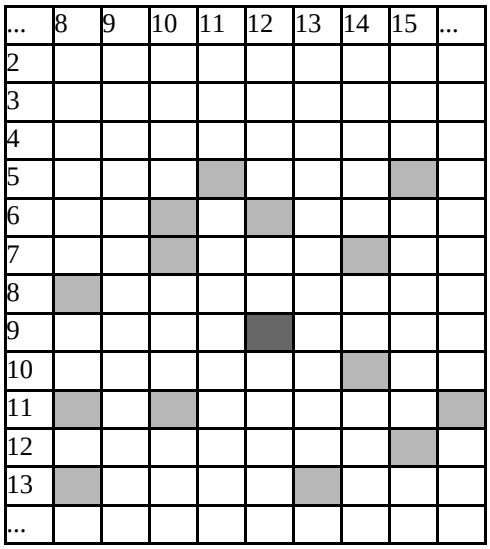


Figure 6C Here the entity has moved four coordinates down. Because the entity is moving faster, the radius has increased. The proportion of coordinates within the radius selected for the encoding is smaller because we are selecting the same number of bits, 15, out of a larger pool of 81 coordinates. Despite the new position being reasonably far away from Y and selecting a sparse set of the coordinates to include in the encoding, the weighting scheme ensures that there are still a handful of coordinates selected for the encodings of both Y and Z.

Example 4 – Encoding Natural Language

Words, sentences, and documents can be encoded into SDRs as well. There are many existing techniques for creating vector encodings of language, including one created by Cortical.io that produces SDRs. Their technique is described in detail in their recent white paper (Webber, 2015).

Choosing the Size of a New Encoder

Every encoder, no matter what it is representing, should create SDRs that have a fixed number of bits n and a fixed number of one bits w. How do you know what are good values for n and w? To maintain the properties that come from sparsity, w can't be a large fraction of n. But if w gets too small then we lose the properties that come from having a distributed representation. As a general rule, for numerical values w should be at least 20 to properly handle noise and subsampling and n should be at least 100 to provide enough resolution to distinguish between many numbers.

Here are some values that have been used successfully in practical applications.

- In one experiment looking at numeric metric values from servers, an optimization algorithm chose n = 134 and w = 21 as good values.
- In another experiment n = 2048 and w = 41 were found to be good values for a geospatial encoder.
- When encoding categories, w can be a higher proportion of n. For example, at one extreme if you encoded a binary value n could be 100 and w could be 50.

When creating a new encoder it is usually a good strategy to pick initially n and w using the broad guidelines. After the encoder is debugged and there are accuracy tests in place, then performance of the encoder can be tweaked by more carefully selecting n and w. A lot of the background for picking good values for n and w came from the work done by Ahmad and Hawkins on the properties of SDRs (Ahmad and Hawkins, 2016).

Encoding Multiple Values

Some applications require multiple values to be encoded for a single HTM model. The separate values can be encoded on their own and then concatenated to form the combined encoding. When doing this, it is important to keep the number of one bits, w, relatively similar for each of the individual value encoders so that one of the values does not dominate the representation. It is fine to have very different values for n, the number of total bits, for the individual encoders.

Conclusion

There are a number of encoders available that should cover the needs for most applications. And if you need to build an encoder for a new data type, there are a few simple rules that you can use to create the encoder:

1. Semantically similar data should result in SDRs with overlapping active bits.
2. The same input should always produce the same SDR as output.
3. The output should have the same dimensionality (total number of bits) for all inputs.
4. The output should have similar sparsity for all inputs and have enough one-bits to handle noise and subsampling.

References

Hawkins, J. & Ahmad, S. (2016). Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex. Frontiers in Neural Circuits, March 2016, Volume 10. Retrieved from http://dx.doi.org/10.3389/fncir.2016.00023

Ahmad, S., & Hawkins, J. (2016). How do neurons operate on sparse distributed representations? A mathematical theory of sparsity, neurons and active dendrites. *arXiv*, 1601.00720. Neurons and Cognition; Artificial Intelligence. Retrieved from http://arxiv.org/abs/1601.00720

Webster D, Popper A, Fay R editors (1992). The Mammalian Auditory Pathway: Neuroanatomy. Springer handbook of auditory research, v1. Springer-Verlag New York, Inc

Schuknecht, H.F. (1974). Pathology of the Ear. Cambridge, MA: Harvard University Press

Webber, F. D. S. (2015). Semantic Folding Theory And its Application in Semantic Fingerprinting, 57. Artificial Intelligence; Computation and Language; Neurons and Cognition. Retrieved from http://arxiv.org/abs/1511.08855

Spatial Pooling Algorithm

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	S. Ahmad
0.5	Feb 2017	Update to current algorithms	M. Taylor & Y. Cui
0.56	Jan 2024	Corrected description of boostFunction(c)	C. Lai

Important Note to Readers:

The following text gives details of the Spatial Pooling algorithm, including pseudocode and parameters. We highly recommend that you access some of the other Spatial Pooling resources available in order to understand the high-level concepts and role of Spatial Pooling in biology, and in HTM. You can find links to the latest Spatial Pooling resources in the Spatial Pooling chapter.

Spatial Pooling Algorithm Details

We first present some important terms, then the high-level steps, followed by details with pseudocode.

Terminology

- Column: An HTM region is organized in columns of cells. The SP operates at the column-level, where a column of a cells function as a single computational unit.
- Mini-column: See “Column”
- Inhibition: The mechanism for maintaining sparse activations of neurons. In the SP this manifests as columns inhibiting nearby columns from becoming active.
- Inhibition radius: The size of a column’s local neighborhood, within which columns may inhibit each other from becoming active.
- Active duty cycle: A moving average denoting the frequency of column activation.
- Overlap duty cycle: A moving average denoting the frequency of the column’s overlap value being at least equal to the proximal segment activation threshold.
- Receptive field: The input space that a column can potentially connect to.
- Permanence value: indicates the amount of growth between a mini-column in the Spatial Pooling algorithm and one of the cells in its receptive field
- Permanence threshold: If a synapse’s permanence is above this value, it is considered fully connected. Acceptable values are [0,1].
- Synapse: A junction between cells. In the Spatial Pooling algorithm, synapses on a column’s dendritic segment connect to bits in the input space. A synapse can be in the following states:
    - Connected—permanence is above the threshold.
    - Potential—permanence is below the threshold.
    - Unconnected—does not have the ability to connect.

Spatial Pooling algorithm steps

1. Start with an input consisting of a fixed number of bits. These bits might represent sensory data or they might come from another region elsewhere in the HTM system.
2. Initialize the HTM region by assigning a fixed number of columns to the region receiving this input. Each column has an associated dendritic segment, serving as the connection to the input space. Each dendrite segment has a set of potential synapses representing a (random) subset of the input bits. Each potential synapse has a permanence value. These values are randomly initialized around the permanence threshold. Based on their permanence values, some of the potential synapses will already be connected; the permanences are greater than than the threshold value.
3. For any given input, determine how many connected synapses on each column are connected to active (ON) input bits. These are active synapses.
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. A small percentage of columns within the inhibition radius with the highest activations (after boosting) become active, and disable the other columns within the radius. The inhibition radius is itself dynamically determined by the spread of input bits. There is now a sparse set of active columns.
6. The region now follows the Spatial Pooling (Hebbian-style) learning rule: 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 connected to unconnected, and vice-versa.
7. For subsequent inputs, we repeat from step 3.

Spatial Pooling Pseudocode

This section contains the detailed pseudocode for the Spatial Pooling function, broken down into four phases: initialization, overlap computation, inhibition, and learning. After initialization (phase 1), every iteration of the Spatial Pooling algorithm’s compute routine goes through three distinct phases (phase 2 through phase 4) that occur in sequence.

The various data structures and supporting routines used in the code are defined in Table X at the end.

Phase 1 – Initialize Spatial Pooling algorithm parameters

Prior to receiving any inputs, the Spatial Pooling algorithm 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 (within a column’s inhibition radius). 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, so that they have higher values near the center.

Phase 2 – Compute the overlap with the current input for each column

Given an input vector, this 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 the column’s boost factor.

```
1. for c in columns
2.     overlap(c) = 0
3.     for s in connectedSynapses(c)
4.         overlap(c) = overlap(c) + input(t, s.sourceInput)
5.     overlap(c) = overlap(c) * boost(c)
```

Phase 3 – Compute the winning columns after inhibition

The third phase calculates which columns remain as winners after the inhibition step. localAreaDensity is a parameter that controls the desired density of active columns within a local inhibition area. Alternatively, the density can be controlled by parameter numActiveColumnsPerInhArea. When using this method, the localAreaDensity parameter must be less than 0. The inhibition logic will ensure that at most numActiveColumnsPerInhArea columns become active in each local inhibition area. For example, if numActiveColumnsPerInhArea is 10, a column will be a winner if it has a non-zero overlap and its overlap score ranks 10th or higher among the columns within its inhibition radius.

```
6.  for c in columns
7.      minLocalActivity = kthScore(neighbors(c), numActiveColumnsPerInhArea)
8.      if overlap(c) > stimulusThreshold and
9.         overlap(c) ≥ minLocalActivity then
10.         activeColumns(t).append(c)
```

Phase 4 – Update synapse permanences and internal variables

This final phase performs learning, updating the permanence values of all synapses as necessary, as well as the boost values and inhibition radii. The main learning rule is implemented in lines 14-20. 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. Notice that permanence values on synapses of non-winning columns are not modified.

Lines 21-27 implement boosting. There are two separate mechanisms in place to help a column learn connections. If a column does not win often enough (as measured by activeDutyCycle) compared to its neighbors, its overall boost value is set to be greater than 1 (line 22-23). If a column is active more frequently than its neighbors, its overall boost value is set to be less than one. The boostFunction is an exponential function that depends on the difference between the active duty cycle of a column and the average active duty cycles of its neighbors. 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 24-27). Note that once learning is turned off, boost(c) is frozen.

Finally, at the end of Phase 4 the inhibition radius is recomputed (line 28).

```
11.for c in activeColumns(t)
12.     for s in potentialSynapses(c)
13.         if active(s) then
14.             s.permanence += synPermActiveInc
15.             s.permanence = min(1.0, s.permanence)
16.         else
17.             s.permanence -= synPermInactiveDec
18.             s.permanence = max(0.0, s.permanence)
19.for c in columns:
20.     activeDutyCycle(c) = updateActiveDutyCycle(c)
21.     activeDutyCycleNeighbors = mean(activeDutyCycle(neighbors(c))
22.     boost(c) = boostFunction(activeDutyCycle(c), activeDutyCycleNeighbors)
23.     overlapDutyCycle(c) = updateOverlapDutyCycle(c)
24.     if overlapDutyCycle(c) < minDutyCycle(c) then
25.         increasePermanences(c, 0.1*connectedPerm)
26.inhibitionRadius = averageReceptiveFieldSize()
```

Algorithm Parameters

The Spatial Pooling algorithm has many parameters that affect the dimensionality, learning mechanisms, and overall performance. Here we discuss some parameters in detail, and how various values influence Spatial Pooling. The roles of the parameters were previously described in the pseudocode, and a full list of the algorithm parameters, data structures, and routines can be found in Tables 1 and 2 below.

Spatial Pooling Algorithm Structure

The column dimensions (columnDimensions) as specified in the Spatial Pooling algorithm parameters define the dimensions for the HTM region. If we allocate 4096 columns, the region is an array of 4096 columns. However, for a vision system, the same

number of columns could be used as a two-dimensional array, so the region's columnar structure would be 64x64. We can also specify a three-dimensional topology.

While the column dimensions control the output shape, the inputs to the columns are controlled by specifying the Spatial Pooling algorithm's input parameters. The input dimensions (inputDimensions) are specified just like the column dimensions, and the number of dimensions must match. The input space that a column can potentially connect to—i.e., the receptive field of the column—is controlled with the potential radius (potentialRadius) parameter. This value will determine the spread of a column's influence across the HTM layer. A small potential radius will keep a column's receptive field local, while a very large potential radius will give the column global coverage over the input space.

Inhibition

With global inhibition (globalInhibition=True), the most active columns are selected from the entire layer. Otherwise the winning columns are selected with respect to the columns' local neighborhoods. The former offers a significant performance boost, and is often what we use in practice. With global inhibition turned off, the columnar inhibition takes effect in local neighborhoods. Column neighborhoods are a function of the inhibition radius (inhibitionRadius), a dynamically calculated measure internal to the Spatial Pooling algorithm that is a function of the average size of the connected receptive fields of all columns. The receptive field of columns can be controlled in part by the potential radius parameter above; it cannot be set explicitly because the receptive fields of Spatial Pooling algorithm columns (and HTM cells in general) are dynamic.

We can however specify the density of active columns, with either localAreaDensity or numActiveColumnsPerInhArea. During inhibition these parameters will be used to calculate the maximum number of columns to remain ON within a local inhibition area. With the former parameter, you specify a density, so the actual number of active columns will change as columns' change the sizes of their receptive fields. Using the latter parameter, the density will fluctuate as the receptive fields change, but the max number of active columns remains fixed.

Learning

The learning rate can be specified with the synapse permanence increment and decrement amounts – typically the former is larger than the latter.

Column Activity

Although a column may win out in competition, its activation must be greater than a threshold (stimulusThreshold) in order to become active. The intent here is to prevent noise from activating columns, which is helpful towards the spatial pooling goal of avoiding trivial patterns.

Boosting can be helpful in driving columns to compete for activation. Boosting is monitored by both the activity and overlap duty cycles (activeDutyCycle(c) and overlapDutyCycle(c), respectively). Following inhibition, if a column's active duty cycle falls below the active duty cycles of neighboring columns, then its internal boost factor (boost(c)) will increase above one. If a column's active duty cycle arises above the active duty cycles of neighboring columns, its boost factor will decrease below one This helps drive the competition amongst columns and achieve the spatial pooling goal of using all the columns. Before inhibition, if a column's overlap duty cycle is below its minimum acceptable value (calculated dynamically as a function of minPctOverlapDutyCycle and the overlap duty cycle of neighboring columns), then all its permanence values are boosted by the increment amount. A subpar duty cycle implies either a column's previously learned inputs are no longer ever active, or the vast majority of them have been "hijacked" by other columns. By raising all synapse permanences in response to a subpar duty cycle before inhibition, we enable a column to search for new inputs.

Parameters, Data Structures, and Routines

The following tables summarize the Spatial Pooling algorithm data structures, routines, and parameters, including recommended parameter settings for typical use cases.

Columns	List of all columns.
columnCount	The total number of columns in the Spatial Pooling algorithm, and the HTM region. *This is task dependent but we recommend a minimum value of 2048.*
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 Pooling algorithm overlap of column c with a particular input pattern.
activeColumns(*t*)	List of column indices that are winners due to bottom-up input.
numActiveColumnsPerInhArea	A parameter controlling the number of columns that will be winners after the inhibition step. *We usually set this to be 2% of the expected inhibition radius. For 2048 columns and global inhibition, this is set to 40. We recommend a minimum value of 25.*
inhibitionRadius	Average connected receptive field size of the columns.
neighbors(c)	A list of all the columns that are within inhibitionRadius of column c.
stimulusThreshold	A minimum number of inputs that must be active for a column to be considered during the inhibition step. *This is roughly the background noise level expected out of the encoder and is often set to a very low value (0 to 5). The system is not very sensitive to this parameter; set to 0 if unsure.*
boost(c)	The boost value for column c as computed during learning – used to increase the overlap value for inactive columns.
boostStrength	A number greater or equal than 0.0 to control the strength of boosting. No boosting is applied if boostStrength=0.
synapse	A data structure representing a synapse, containing a permanence value and the source input index.
potentialPct	The percent of the inputs, within a column's potential radius, that are initialized to be in this column's potential synapses. *This should be set so that on average, at least 15-20 input bits are connected when the Spatial Pooling algorithm is initialized. For example, suppose the input to a column typically contains 40 ON bits and that permanences are initialized such that 50% of the synapses are initially connected. In this case you will want potentialPct to be at least 0.75 since 40*0.5*0.75 = 15.*
connectedPerm	If the permanence value for a synapse is greater than this value, it is said to be connected. *This is usually set to 0.2. The Spatial Pooling algorithm is not sensitive to this parameter.*
potentialSynapses(c)	The list of potential synapses and their permanence values for this column.

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.
synPermActiveInc	Amount permanence values of active synapses are incremented during learning. *This parameter is somewhat data dependent. (The amount of noise in the data will determine the optimal ratio between synPermActiveInc and synPermInactiveDec.) Usually set to a small value, such as 0.03*
synPermInactiveDec	Amount permanence values of inactive synapses are decremented during learning. *Usually set to a value smaller than the increment, such as 0.015.*
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 stimulusThreshold) with its inputs (e.g. over the last 1000 iterations).

Table 1. Variables and data structures used in the Spatial Pooling pseudocode and NuPIC implementation. For the parameters we include settings that work well under a wide range of scenarios (italicized above).

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 stimulusThreshold.
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.
active(s)	True if synapse s is active, i.e. the input connected to synapse s is ON.
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 positive scalar value. It is above one if the active duty cycle is below the mean active duty cycles of neighboring columns. It is less than one if a column has higher active duty cycle than its neighbors

Table 2. Supporting routines used in the Spatial Pooling pseudocode. They may have different names in the NuPIC codebase.

Temporal Memory Algorithm

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	S. Ahmad
0.5	Feb 2017	Update to current algorithms	M. Lewis
0.56	May 2021	Updated figures in Temporal Memory Algorithm to reflect current thinking on Thousand Brains Theory of Intelligence and made some clarifications	C. Lai

Important Note to Readers:

The following text gives details of the Temporal Memory algorithm, including pseudocode and parameters. We highly recommend that you read about the Spatial Pooling algorithm first, and access some of the other Temporal Memory resources available in order to understand the high-level concepts and role of Temporal Memory in biology, and in HTM. You can find links to the latest Temporal Memory resources in the Temporal Memory chapter.

Temporal Memory Algorithm Details

The Temporal Memory algorithm does two things: it learns sequences of Sparse Distributed Representations (SDRs) formed by the Spatial Pooling algorithm, and it makes predictions. Specifically, the Temporal Memory algorithm:

1) Forms a representation of the sparse input that captures the temporal context of previous inputs
2) Forms a prediction based on the current input in the context of previous inputs

We will discuss each of these steps in more detail, but first we present some important terms and concepts.

Terminology

- Column: An HTM region is organized in columns of cells.
- Mini-column: See “Column”.
- Permanence value: A scalar value (0.0 to 1.0) that is assigned to each synapse to indicate how permanent the connection is. When a connection is reinforced, its permanence value is increased. Under other conditions, the permanence value is decreased.
- Permanence threshold: If a synapse’s permanence value is above this threshold, it is considered fully connected. Acceptable values are [0,1].
- Synapse: A junction between cells. A synapse can be in the following states:
    - Connected—permanence is above the threshold.
    - Potential—permanence is below the threshold.
    - Unconnected—does not have the ability to connect.

Concepts Shared Between the Spatial Pooling and Temporal Memory Algorithms

Binary weights

HTM synapses have only a 0 or 1 effect on the post-synaptic cell; their "weight" is binary, unlike many neural network models that use scalar variable values in the range of 0.0 to 1.0.

Dendrite segments

Synapses connect to dendrite segments. There are two types of dendrite segments, proximal and distal.

- A proximal 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.

- A distal dendrite segment forms synapses with cells within the layer. Every cell has many distal dendrite segments. If the sum of the active synapses on a distal segment exceeds a threshold, then the associated cell enters the predicted state. Since there are multiple distal dendrite segments per cell, a cell's predictive state is the logical OR operation of several constituent threshold detectors.

Potential Synapses

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 the permanence threshold.

Learning

Learning in the Spatial Pooling and Temporal Memory algorithms are similar: in both cases learning involves establishing connections, or synapses, between cells. The Temporal Memory algorithm described here learns connections between cells in the same layer. The Spatial Pooling algorithm learns feed-forward connections between input bits and columns.

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 Pooling and Temporal Memory algorithms. The details for the Temporal Memory algorithm are described below.

Now we will discuss concepts specific to the Temporal Memory algorithm.

Temporal Memory Algorithm Concepts

The Temporal Memory algorithm learns sequences and makes predictions. In the Temporal Memory algorithm, 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.

Use of Sparse Distributed Representation Properties

It is worth noting a few important properties of Sparse Distributed Representations (SDRs) that the Temporal Memory algorithm exploits.

Assume we have a hypothetical layer 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 and we have reduced our memory needs considerably.

The cells in an HTM layer take advantage of this property. Each of a cell's dendrite segments has a set of connections to other cells in the layer. 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 multiple 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 then set a threshold so the dendrite segment becomes active when any 15 of its connections are active. This introduces the possibility for error. It is possible, by chance, that the dendrite reaches its threshold of 15 active connections by mixing parts of different patterns. 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.

First Order Versus Variable Order Sequences and Prediction

What is the effect of having more or fewer cells per column? Specifically, what happens if we have only one cell per column?

As we show later in this chapter, a representation of an input comprised of 100 active columns with 4 cells per column can be encoded in 4^100 different ways. Therefore, the same input can appear in a many contexts without confusion. For example, if input patterns represent words, then a layer can remember many sentences that use the same words over and over again and not get confused. A word such as "dog" would have a unique representation in different contexts. This ability permits an HTM layer to make what are called "variable order" predictions. A variable order prediction is not based solely on what is currently happening, but on varying amounts of past context. An HTM layer 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. In practice we have found using 10 to 16 cells per column to be sufficient for most situations.

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 layer always results in the same prediction, regardless of the context. With one cell per column, the memory of an HTM layer 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 layer with multiple cells per column is ideally suited for recognizing time-based sequences, and an HTM layer with one cell per column is ideally suited to recognizing spatial patterns.

Temporal Memory Algorithm Steps

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

The input to the Temporal Memory algorithm is sequences of Sparse Distributed Representations (SDRs) that are formed by the Spatial Pooling algorithm. As described in the "Spatial Pooling Algorithm" chapter of this book, the Spatial Pooling algorithm forms a sparse distributed representation (SDR) of the input to a layer. This SDR represents columns of cells that received the most input (Fig. 1).

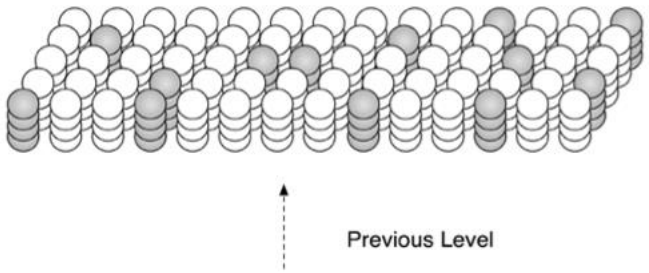


Figure 1 Depiction of the Spatial Pooling algorithm. An HTM layer consists of columns of cells. Only a small portion of a layer is shown. Each column of cells receives activation from a unique subset of the input. Columns with the strongest activation inhibit columns with weaker activation. The result is a sparse distributed representation of the input. The figure shows active columns in light grey. (When there is no prior state, every cell in the active columns will be active, as shown.)

After Spatial Pooling, the Temporal Memory algorithm converts 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 (Fig. 2).

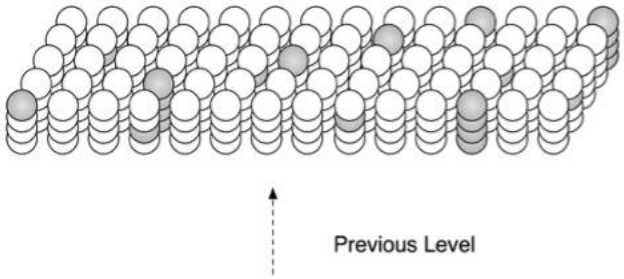


Figure 2 By activating a subset of cells in each column, an HTM layer can represent the same input in many different contexts. Columns only activate predicted cells. Columns with no predicted cells activate all the cells in the column. The figure shows some columns with one cell active and some columns with all cells active.

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".

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 layer. It is hard to overemphasize the importance of this capability.

Each column in an HTM layer consists of multiple cells. All cells in a column get the same feed-forward input. 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. For example, 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 layer is the following. When a column becomes active, it looks at all the cells in the column. If one or more cells in the column are already in the predictive state, only those cells become active. If no cells in the column are in the predictive state, then all the cells become active. The system essentially confirms its expectation when the input pattern matches, by only activating the cells in the predictive state. If the input pattern is unexpected, then the system activates all cells in the column as if to say that all possible interpretations are valid.

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.

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 partially depolarized due to lateral connections to other nearby cells, we say it is in the "predictive state" (Fig. 3).

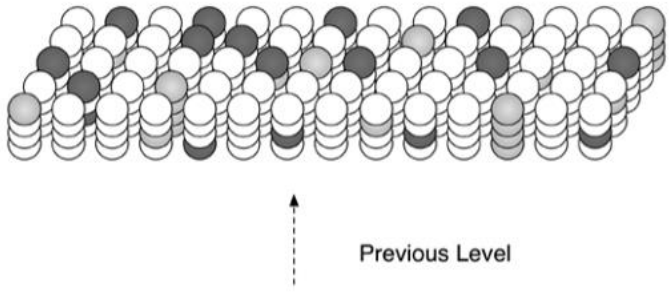


Figure 3 At any point in time, some cells in an HTM layer will be active due to feed-forward input (shown in light gray). Other cells that receive lateral input from active cells will be in a predictive state (shown in dark gray).

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

The next step for the Temporal Memory algorithm 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 layer makes a prediction it depolarizes (i.e. puts into the predictive state) all the cells that will likely become active due to future feed-forward input. Because representations in a layer 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 ten cells, with one predicted at a time, then 4% of the cells would be in the predictive state.

The chapter on Sparse Distributed Representations shows that even though different predictions are merged (union'ed) together, an HTM layer can know with high certainty whether a particular input was predicted or not.

How does a layer 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 state for other sets of cells that typically follow. Think of this as the moment when you recognize a song and start predicting the next notes.

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, and these in turn cause other cells to enter a predictive state through learned connections between cells in the layer. The cells that are depolarized by connections within the layer 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, then 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.

Temporal Memory Algorithm Pseudocode

The Temporal Memory algorithm starts where the Spatial Pooling algorithm leaves off, with a set of active columns representing the feed-forward input. In the pseudocode below, a time step consists of the following computations:

1. Receive a set of active columns, evaluate them against predictions, and choose a set of active cells:
    a. For each active column, check for cells in the column that have an active distal dendrite segment (i.e. cells that are in the "predictive state" from the previous time step), and activate them. If no cells have active segments, activate all the cells in the column, marking this column as "bursting". The resulting set of active cells is the representation of the input in the context of prior input.
    b. For each active column, learn on at least one distal segment. For every bursting column, choose a segment that had some active synapses at any permanence level. If there is no such segment, grow a new segment on the cell with the fewest segments, breaking ties randomly. On each of these learning segments, increase the permanence on every active synapse, decrease the permanence on every inactive synapse, and grow new synapses to cells that were previously active.

2. Activate a set of dendrite segments: for every dendrite segment on every cell in the layer, count how many connected synapses correspond to currently active cells (computed in step 1). If the number exceeds a threshold, that dendrite segment is marked as active. The collection of cells with active distal dendrite segments will be the predicted cells in the next time step.

At the end of each time step, we increment the time step counter "t".

Rather than storing a set of predicted cells, the algorithm stores a set of active distal dendritic segments. A cell is predicted if it has an active segment.

1) Evaluate the active columns against predictions. Choose a set of active cells.

Translate a set of active columns into a set of active cells, and grow / reinforce / punish distal synapses to better predict these columns.

First, for each column, decide whether it is any of the following:

- Active and has one or more predicted cells
- Active and has no predicted cells
- Inactive and has at least one predicted cell

Handle each case in a different function, defined further below.

```
1. for column in columns
2.     if column in activeColumns(t) then
3.         if count(segmentsForColumn(column, activeSegments(t-1))) > 0 then
4.             activatePredictedColumn(column)
5.          else
6.             burstColumn(column)
7.     else
8.         if count(segmentsForColumn(column, matchingSegments(t-1))) > 0 then
9.            punishPredictedColumn(column)
```

A column is predicted if any of its cells have an active distal dendrite segment (lines 3-4). If none of the column's cells have an active segment, the column bursts (line 6).

Dendrite segments in inactive columns are punished if they are active enough to be "matching" (lines 8-9).

```
10. function activatePredictedColumn(column)
11.     for segment in segmentsForColumn(column, activeSegments(t-1))
12.         activeCells(t).add(segment.cell)
13.         winnerCells(t).add(segment.cell)
14.
15.         if LEARNING_ENABLED:
16.             for synapse in segment.synapses
17.                 if synapse.presynapticCell in activeCells(t-1) then
18.                     synapse.permanence += PERMANENCE_INCREMENT
19.                 else
20.                     synapse.permanence -= PERMANENCE_DECREMENT
21.
22.             newSynapseCount = (SYNAPSE_SAMPLE_SIZE -
23.
   numActivePotentialSynapses(t-1, segment))
24.             growSynapses(segment, newSynapseCount)
```

For each active column, if any cell was predicted, those predicted cells become active (lines 11-12). Each of these cells is marked as a "winner" (line 13), making them presynaptic candidates for synapse growth in the next time step.

For each of these correctly active segments, reinforce the synapses that activated the segment, and punish the synapses that didn't contribute (lines 16-20). If the segment has fewer than SYNAPSE_SAMPLE_SIZE active synapses, grow new synapses to a subset of the winner cells from the previous time step to make up the difference (lines 22 – 24).

```
25. function burstColumn(column)
26.     for cell in column.cells
27.         activeCells(t).add(cell)
28.
29.     if segmentsForColumn(column, matchingSegments(t-1)).length > 0 then
30.         learningSegment = bestMatchingSegment(column)
31.         winnerCell = learningSegment.cell
32.     else
33.         winnerCell = leastUsedCell(column)
34.         if LEARNING_ENABLED:
35.             learningSegment = growNewSegment(winnerCell)
36.
37.     winnerCells(t).add(winnerCell)
38.
39.     if LEARNING_ENABLED:
40.         for synapse in learningSegment.synapses
41.             if synapse.presynapticCell in activeCells(t-1) then
42.                 synapse.permanence += PERMANENCE_INCREMENT
43.             else
44.                 synapse.permanence -= PERMANENCE_DECREMENT
45.
46.         newSynapseCount = (SAMPLE_SIZE -
47.                                             numActivePotentialSynapses(t-1,
    learningSegment))
48.         growSynapses(learningSegment, newSynapseCount)
```

If the column activation was unexpected, then each cell in the column becomes active (lines 26-27).

Select a winner cell and a learning segment for the column (lines 29-35). If any cells have a matching segment, select the best matching segment and its cell (lines 30-31). Otherwise select the least used cell and grow a new segment on it (line 33-35).

On the learning segment, reinforce the synapses that partially activated the segment, and punish the synapses that didn't contribute (lines 40-44). Then grow new synapses to a subset of the previous time step's winner cells (lines 46-48).

```
49. function punishPredictedColumn(column)
50.     if LEARNING_ENABLED:
51.         for segment in segmentsForColumn(column, matchingSegments(t-1))
52.             for synapse in segment.synapses
53.                 if synapse.presynapticCell in activeCells(t-1) then
54.                     synapse.permanence -= PREDICTED_DECREMENT
```

When a column with matching segments doesn't become active, punish the synapses that caused these segments to be "matching".

2) Activate a set of dendrite segments.

Calculate a set of active and matching dendrite segments. Active segments denote predicted cells. Matching segments are used in the next time step when choosing which cells to learn on.

```
55. for segment in segments
56.     numActiveConnected = 0
57.     numActivePotential = 0
58.     for synapse in segment.synapses
59.         if synapse.presynapticCell in activeCells(t) then
60.             if synapse.permanence ≥ CONNECTED_PERMANENCE then
61.                 numActiveConnected += 1
62.
63.             if synapse.permanence ≥ 0 then
64.                 numActivePotential += 1
65.
66.     if numActiveConnected ≥ ACTIVATION_THRESHOLD then
67.         activeSegments(t).add(segment)
68.
69.     if numActivePotential ≥ LEARNING_THRESHOLD then
70.         matchingSegments(t).add(segment)
71.
72.     numActivePotentialSynapses(t, segment) = numActivePotential
```

A synapse is “connected“ if its permanence is sufficiently high (line 60). For each segment, count the connected and potential synapses that are active (lines 58-64). If either number is high enough, the segment is considered “active” or “matching”, respectively (lines 66-70).

Record the number of active potential synapses so that we can use it during learning in the next time step (line 72, used in lines 23, 47).

Now, repeat. Go back to Stage 1. Putting these two stages together, you get the entire algorithm.

Supporting functions

These functions are used above.

```
73. function leastUsedCell(column)
74.     fewestSegments = INT_MAX
75.     for cell in column.cells
76.         fewestSegments = min(fewestSegments, cell.segments.length)
77.
78.     leastUsedCells = []
79.     for cell in column.cells
80.         if cell.segments.length == fewestSegments then
81.             leastUsedCells.add(cell)
82.
83.     return chooseRandom(leastUsedCells)
```

The least used cell is the cell with the fewest segments. Find all the cells with the fewest segments (lines 74-81) and choose one randomly (line 83).

```
84. function bestMatchingSegment(column)
85.     bestMatchingSegment = None
86.     bestScore = -1
87.     for segment in segmentsForColumn(column, matchingSegments(t-1))
88.         if numActivePotentialSynapses(t-1, segment) > bestScore then
89.             bestMatchingSegment = segment
90.             bestScore = numActivePotentialSynapses(t-1, segment)
91.
92.     return bestMatchingSegment
```

The best matching segment is the “matching” segment with largest number of active potential synapses.

```
93. function growSynapses(segment, newSynapseCount)
94.     candidates = copy(winnerCells(t-1))
95.     while candidates.length > 0 and newSynapseCount > 0
96.         presynapticCell = chooseRandom(candidates)
97.         candidates.remove(presynapticCell)
98.
99.         alreadyConnected = false
100.       for synapse in segment.synapses
101.           if synapse.presynapticCell == presynapticCell then
102.               alreadyConnected = true
103.
104.       if alreadyConnected == false then
105.           newSynapse = createNewSynapse(segment, presynapticCell,
106.
   INITIAL_PERMANENCE)
107.           newSynapseCount -= 1
```

When growing synapses, grow to a random subset of the previous time step’s winner cells. A segment can only connect to a presynaptic cell once (lines 99-104).

Temporal Memory Implementation, Data Structures and Routines

In this section we describe some of the details of our Temporal Memory implementation and terminology.

Each Temporal Memory cell has a list of dendrite segments, where each segment contains a list of synapses. Each synapse has a presynaptic cell and a permanence value.

The implementation of potential synapses is different from the implementation in the Spatial Pooling algorithm. In the Spatial Pooling algorithm, the complete list of potential synapses is represented as an explicit list. In the Temporal Memory algorithm, 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 Memory algorithm, we randomly add active synapses to each segment during learning (controlled by the parameter SYNAPSE_SAMPLE_SIZE). 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 Temporal Memory algorithm allows three mutually exclusive cell states: “active”, “inactive”, and “predictive”. But in the pseudocode above, it is possible for a cell to be both active and predictive. This happens occasionally, for example with the sequence “AABC”, if you burst “A”, the Temporal Memory algorithm will predict “A” and “B”, so at this point some “A” cells are both active and predictive.

The Temporal Memory algorithm allows a single distal dendrite segment to form connections to multiple patterns of previously active cells, but the pseudocode above generally does not do this. It learns one pattern per segment. This approach grows more dendrite segments than is strictly necessary, but this doesn’t have a fundamental impact on the algorithm.

t	The current time step. This is incremented after the “Activate Dendrites” phase.
columns	A list of all columns
segments	A list of all segments
activeColumns(t)	Set of active columns at time t. This is the input to the Temporal Memory.
activeCells(t)	Set of active cells at time t
winnerCells(t)	Set of winner cells at time t
activeSegments(t)	Set of active segments at time t
matchingSegments(t)	Set of matching segments at time t
numActivePotentialSynapses (t, segment)	The number of active potential synapses on the given segment at time t. A “potential synapse” is a synapse with any permanence – it doesn’t have to reach the CONNECTED_PERMANENCE.
ACTIVATION_THRESHOLD	Activation threshold for a segment. If the number of active connected synapses in a segment is ≥ this value, the segment is “active”.
INITIAL_PERMANENCE	Initial permanence value for a synapse.
CONNECTED_PERMANENCE	If the permanence value for a synapse is ≥ this value, it is “connected”.
LEARNING_THRESHOLD	Learning threshold for a segment. If the number of active potential synapses in a segment is ≥ this value, the segment is “matching”, and it is qualified to grow and reinforce synapses to the previous active cells.
LEARNING_ENABLED	Either True or False. If True, the Temporal Memory should grow / reinforce / punish

	synapses each timestep.
PERMANENCE_INCREMENT	If a segment correctly predicted a cell's activity, the permanence values of its active synapses are incremented by this amount.
PERMANENCE_DECREMENT	If a segment correctly predicted a cell's activity, the permanence values of its inactive synapses are decremented by this amount.
PREDICTED_DECREMENT	If a segment incorrectly predicted a cell's activity, the permanence values of its active synapses are decremented by this amount.
SYNAPSE_SAMPLE_SIZE	The desired number of active synapses on a segment. A learning segment will grow synapses to reach this number. The segment connects to a subset of an SDR, and this is the size of that subset.

Table 1. Variables and data structures used in the Temporal Memory pseudocode.

segmentsForColumn(column, segments)	Given a set of segments, return the segments that are on a cell in this column.
count(collection)	Get the number of elements in this collection.
chooseRandom(collection)	Return a random element from this collection.
collection.add(element)	Add an element to this collection. Don't allow duplicates
collection.remove(element)	Remove an element from this collection.
chooseRandom(collection)	Return a random element from this collection.
min(a, b)	Equivalent to "if a ≤ b then return a else return b"

Table 2. Supporting routines used in the Temporal Memory pseudocode. They may have different names in the NuPIC codebase.

Voting Across Columns

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.5	Nov 2019	Initial release	C. Maver

Voting Across Columns

In this chapter, we introduce a core principle of the Thousand Brains Theory of Intelligence: voting across columns. This concept was published in a peer-reviewed paper, "A Theory of How Columns in the Neocortex Enable Learning the Structure of the World," in the journal *Frontiers in Neural Circuits.* The paper also has an accompanying video. You can find these resources, along with FAQs here.

Object Recognition: Learning through movement

We have proposed a network model that learns the structure of objects through movement. Our model is based on the known biology of cortical columns and layers, and helps explain their function. Object recognition in our model can be achieved in two ways:

- Over time, as individual columns integrate changing inputs to recognize complete objects
- Through voting across columns. Existing lateral connections allow multiple columns to integrate inputs over space, thereby inferring more quickly, based on shared partial knowledge among adjacent columns.

Location Layer in Grid Cells

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.5	Nov 2019	Initial release	C. Maver

Location Layer in Grid Cells

In this chapter, we describe the underlying neural mechanism of how the cortex processes a sensorimotor sequence by converting it into a sequence of sensory features at object-centric locations. Our October 2017 paper, "A Theory of How Columns in the Neocortex Enable Learning the Structure of the World," proposed that the cortex processes a sensorimotor sequence. It then learns objects by learning sets of sensory features at locations. However that paper did not provide a neural mechanism for computing object-centric locations. This new paper provides such a neural mechanism. With this missing piece filled in, we present a neural network model that learns to recognize static objects, receiving only a sensorimotor sequence as input.

This concept was published in a peer-reviewed paper, "Locations in the Neocortex: A Theory of Sensorimotor Object Recognition Using Cortical Grid Cells," in the journal *Frontiers in Neural Circuits.* The paper also has an accompanying video. You can find these resources, along with FAQs here.

Related HTM Content

Chapter Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	C. Baranski
0.5	March 2017	Take out references to learning algorithms (separate web pages published)	C. Baranski
0.51	November 2019	Updated terminology	C. Maver

Related HTM Content

Following is a list of additional HTM topics that are not covered in the existing chapters. Where possible, we have included links to current papers or other documentation that offer information on the specific topics.

HTM Neuron

The HTM Neuron includes topics such as synapses, active dendrites, immunity to noise, etc. For more information on the HTM Neuron, you can read the paper "Why Neurons Have Thousands of Synapses, a Theory of Sequence Memory in Neocortex" by J. Hawkins & S. Ahmad (2016). Published in Frontiers in Neural Circuits, March 2016, Volume 10, https://numenta.com/neuroscience-research/research-publications/papers/why-neurons-have-thousands-of-synapses-theory-of-sequence-memory-in-neocortex/

HTM Cellular Layer

The HTM Cellular Layer includes topics such as mini-columns, sequence memory and temporal pooling. For some information on HTM sequence memory, you can read the paper "Continuous Online Sequence Learning with an Unsupervised Neural Network Model" by Y. Cui, S. Ahmad, J. Hawkins & C. Surpur. https://numenta.com/neuroscience-research/research-publications/papers/continuous-online-sequence-learning-with-an-unsupervised-neural-network-model/

HTM Applications

For more information on HTM applications, including our legacy example applications, visit https://numenta.com/machine-intelligence-technology/applications/.

Problem Sets

Problem Sets Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	NA
0.5	May 15 2017	Added questions for SDR, Encoder, SP and TM chapters	S. Purdy

Problem Sets

Sparse Distributed Representations

1. What are the disadvantages to SDRs? Consider representing a string of text as ASCII vs. SDRs.
2. What does the "distributed" part of "Sparse Distributed Representation" mean?
3. Why is sparsity necessary? How sparse do representations need to be? Can they be too sparse?
4. Which is better? n=1000 and w=3 or n=300 and w=15?
5. For representations with 33 active bits out of 500 total, how many unique representations are there? If we randomly generate 100 of these representations, how likely is it that two of the hundred will be identical? If we randomly generate two of these representations, how likely is it that they will share more than 5 active bits?
6. Work out the first few terms of equation (4) in the SDR chapter to see how the values quickly diminish. Show the equation can then be approximated with $fp_w^n(\theta) \approx \frac{|\Omega_x(n,w,\theta)|}{\binom{n}{w}}$.

Encoders

1. When creating an encoder for a new data type, what makes a good SDR? How can you test that the SDRs that are produced are good?
2. How would you encode sound as SDRs?

Spatial Pooling

1. What is Spatial Pooling? How can we tell if the output of Spatial Pooling is good?
2. What is boosting and how does it work? Is it necessary?
3. Is an untrained spatial pooler just a "random hash" (random mapping from input to output vector)? Why or why not? What happens to the output of the spatial pooler if you change one bit in the input?
4. Can untrained spatial poolers with randomly initialized input bit weights be useful or even better than a trained SP?
5. How does online learning happen in the SP?
6. Can you do spatial pooling with small numbers? For example, is it reasonable to have an SP with 20 columns? If not, why are large numbers important in SDR's?
    a. What's the difference between picking "5 columns out of 50" vs "50 out of 500"? Both have 10% sparsity.
    b. What's the difference between picking "50 out of 100" vs "50 out of 1000"? Both will output 50 1's.
7. Is it possible that the SP output for a fixed input can change completely over time? How can this risk be mitigated?
8. Suppose the input vector (input to the SP) is 10,000 bits long, with 5% of the bits active at a time. What percentage of the input space should be in the potential pool for each column? How do you figure this out?
9. How does the SDR representation of input A in isolation, and input B in isolation, compare with the SDR representation of input A unioned with B? Alternatively, how does the representation of a horizontal line and the representation of a vertical line compare with the representation of a cross?
10. Let's look at a Spatial Pooler of 2048 columns with 40 active at a time and learning enabled. A, B, and C represent non-overlapping subsets of 20 bits in the input space (with total of 500 bits). The spatial pooler is trained on the following:
    - All A and B bits active and all other input bits inactive
    - All A and C bits active and all other input bits inactive

- Many other patterns with 40 active bits, but none that overlap significantly with A, B, or C

After this training, consider example inputs when only A bits are active, only B, and only C. How will the overlap compare between the output representations for these three distinct inputs?

11. Suppose we have an input vector that is 10,000 bits long. Suppose the spatial pooler has 500 columns, of which 50 are active at any time.
    a. Can we distinguish many patterns, or a small number? Which patterns are likely to be confused?
    b. What happens to the SDR representation if we add noise to the patterns?
    c. What happens if we add occlusions?

Temporal Memory

1. What is a first order sequence memory? What is a high order sequence memory? What is a variable order sequence memory?
2. Why do we need high order sequence memory?
3. How are sequences learned in the Temporal Memory?
4. What happens when there are no internal predictions and a new set of columns become active?
5. Once a sequence is learned, how does the Temporal Memory make predictions?
6. What is the difference between one cell per column vs multiple cells per column (in terms of what sequences you can learn)?
7. Can the Temporal Memory recognize a sequence if it starts in the middle? How does it do so?
8. How long of a sequence can the Temporal Memory learn? If you have 10,000 inputs into the spatial pooler, 500 columns, and a "reasonable" set of TM parameters (choose some), can you learn a sequence that is 1000 elements long? What about 100,000 elements long?
9. Suppose a Temporal Memory has learned the following two sequences: ABCDE and ABCDF. You now present the sequence ABCD - what will be predicted next? What is the representation of the Temporal Memory at this point in time?
10. Suppose a Temporal Memory has only learned the following two sequences: ABCDE and FGCDH. You now present the sequence FGCD - what will be predicted next? Suppose you present the sequence CD - what will be predicted next? What is the exact representation of the Temporal Memory at this point in time (predicted and active cells).
11. What would be the difference between enforcing one distal segment per cell vs allowing multiple distal segments per cell?
12. What are the disadvantages of the temporal memory? When will other prediction techniques work better?

Glossary

Glossary Revision History

The table notes major changes between revisions. Minor changes such as small clarifications or formatting changes are not noted.

Version	Date	Changes	Principal Author(s)
0.4		Initial release	NA

Glossary

AND: (see *intersection*)

Binary vector: An array of bits. SDRs are represented as binary vectors.

Bit: A single element of an SDR. Can be in either ON (1) or OFF (0) states.

Encoder: Converts the dative format of data into an SDR that can be fed into an HTM system.

False negative: A result that is incorrectly predicted as negative.

False positive: A result that is incorrectly predicted as positive.

Hierarchical Temporal Memory (HTM): A theoretical framework for both biological and machine intelligence.

HTM learning algorithms: Describes the set of algorithms in HTM.

Intersection: Of two sets A and B, the intersection is the set that contains all elements of A that also belong to B, but no other elements; the AND operation, denoted $A \cap B$.

Noise: Meaningless or corrupt data. In SDRs this manifests as randomly flipped ON and OFF bits.

NuPIC: Numenta Platform for Intelligent Computing. An open-source community working on HTM.

OR: (see *union*)

Sparse distributed representation (SDR): Binary representations of data comprised of many bits with a small percentage of the bits active (1's). The bits in these representations have semantic meaning and that meaning is distributed across the bits.

Sparsity: In a binary vector, the ON bits as a percentage of total bits.

Spatial Pooler: One of the HTM learning algorithms. In an HTM region, the Spatial Pooler learns the connections to each column from a subset of the inputs, determines the level of input to each column and uses inhibition to select a sparse set of active columns.

Temporal Memory: Learns sequences of patterns over time, and predicts the next pattern as an SDR at the level of cells in columns.

Temporal Pooler: One of the HTM learning algorithms. The Temporal Pooler groups together SDRs that are predictable by the lower layer, forming a single representation for many different SDRs.

True positive: A result that is correctly predicted as positive.

True negative: A result that is correctly predicted as negative.

Union: The union of two sets A and B is the set of elements which are in A, in B, or in both A and B; the OR operation, denoted $A \cup B$.

Vector cardinality: The number of non-zero elements in a vector, or the l_0-norm.

Vector size: Number of elements in a 1-dimensional vector.