* [Store](https://leanpub.com/bookstore)
* [Read](https://leanpub.com/user_dashboard/library)
* [Write](https://leanpub.com/authors)
* [Support](https://leanpub.com/help)
* [Blog](https://medium.com/@leanpub)
* [Podcasts](https://leanpub.com/podcasts/frontmatter)

Top of Form

SEARCH QUERY

Bottom of Form

[0](https://leanpub.com/shopping_cart?stage=review)

[Sign In](https://leanpub.com/sign_in)[Sign up](https://leanpub.com/sign_up)

**Real Machine Intelligence with Clortex and NuPIC**

[](https://leanpub.com/realsmartmachines)

Real Machine Intelligence with Clortex and NuPIC

[**Buy on Leanpub**](https://leanpub.com/realsmartmachines)

**Table of Contents**

* + [Preface](https://leanpub.com/realsmartmachines/read#leanpub-auto-preface)
    - [Numenta Platform for Intelligent Computing (NuPIC)](https://leanpub.com/realsmartmachines/read#leanpub-auto-numenta-platform-for-intelligent-computing-nupic)
    - [Clortex](https://leanpub.com/realsmartmachines/read#leanpub-auto-clortex)
    - [Credits](https://leanpub.com/realsmartmachines/read#leanpub-auto-credits)
  + [Introduction](https://leanpub.com/realsmartmachines/read#leanpub-auto-introduction)
  + [One: Perspective](https://leanpub.com/realsmartmachines/read#leanpub-auto-one-perspective)
    - [Brain Theory as a Basis for AI](https://leanpub.com/realsmartmachines/read#leanpub-auto-brain-theory-as-a-basis-for-ai)
    - [Machine Intelligence versus Machine Learning](https://leanpub.com/realsmartmachines/read#leanpub-auto-machine-intelligence-versus-machine-learning)
  + [Two: Jeff Hawkins’ Theory of the Neocortex](https://leanpub.com/realsmartmachines/read#leanpub-auto-two-jeff-hawkins-theory-of-the-neocortex)
    - [Hierarchical Temporal Memory and the Cortical Learning Algorithm](https://leanpub.com/realsmartmachines/read#leanpub-auto-hierarchical-temporal-memory-and-the-cortical-learning-algorithm)
    - [A Little Neuroscience Primer](https://leanpub.com/realsmartmachines/read#leanpub-auto-a-little-neuroscience-primer)
    - [Neuroscience in Plain Language](https://leanpub.com/realsmartmachines/read#leanpub-auto-neuroscience-in-plain-language)
  + [Three: The Cortical Learning Algorithm](https://leanpub.com/realsmartmachines/read#leanpub-auto-three-the-cortical-learning-algorithm)
    - [The CLA Model Neuron](https://leanpub.com/realsmartmachines/read#leanpub-auto-the-cla-model-neuron)
    - [The CLA Model of a Neocortical Region](https://leanpub.com/realsmartmachines/read#leanpub-auto-the-cla-model-of-a-neocortical-region)
    - [Sparse Distributed Representations](https://leanpub.com/realsmartmachines/read#leanpub-auto-sparse-distributed-representations)
    - [Creating a Sparse Distributed Representation: Pattern Memory (Spatial Pooling)](https://leanpub.com/realsmartmachines/read#leanpub-auto-creating-a-sparse-distributed-representation-pattern-memory-spatial-pooling)
    - [Prediction Part I - Transition Memory](https://leanpub.com/realsmartmachines/read#leanpub-auto-prediction-part-i---transition-memory)
  + [Four: Numenta Platform for Intelligent Computing (NuPIC)](https://leanpub.com/realsmartmachines/read#leanpub-auto-four-numenta-platform-for-intelligent-computing-nupic)
    - [Turning Theory into Practice](https://leanpub.com/realsmartmachines/read#leanpub-auto-turning-theory-into-practice)
    - [How NuPIC works](https://leanpub.com/realsmartmachines/read#leanpub-auto-how-nupic-works)
    - [The NuPIC Community](https://leanpub.com/realsmartmachines/read#leanpub-auto-the-nupic-community)
    - [NuPIC Architecture](https://leanpub.com/realsmartmachines/read#leanpub-auto-nupic-architecture)
    - [nupic.core](https://leanpub.com/realsmartmachines/read#leanpub-auto-nupiccore)
    - [Some Applications](https://leanpub.com/realsmartmachines/read#leanpub-auto-some-applications)
    - [CEPT - The Cortical Engine for Processing Text](https://leanpub.com/realsmartmachines/read#leanpub-auto-cept---the-cortical-engine-for-processing-text)
  + [Five: Clortex - A Complementary Project](https://leanpub.com/realsmartmachines/read#leanpub-auto-five-clortex---a-complementary-project)
    - [Introduction](https://leanpub.com/realsmartmachines/read#leanpub-auto-introduction-1)
    - [Design Goals](https://leanpub.com/realsmartmachines/read#leanpub-auto-design-goals)
    - [Design Philosophy](https://leanpub.com/realsmartmachines/read#leanpub-auto-design-philosophy)
    - [Platform - Clojure, Functional Programming on the Java Virtual Machine](https://leanpub.com/realsmartmachines/read#leanpub-auto-platform---clojure-functional-programming-on-the-java-virtual-machine)
    - [Architecture - Russ Miles’ Life Preserver](https://leanpub.com/realsmartmachines/read#leanpub-auto-architecture---russ-miles-life-preserver)
  + [What’s Next?](https://leanpub.com/realsmartmachines/read#leanpub-auto-whats-next)
    - [Jeff’s New Theory - Sensorimotor HTM](https://leanpub.com/realsmartmachines/read#leanpub-auto-jeffs-new-theory---sensorimotor-htm)
    - [Third-party Projects Based on HTM](https://leanpub.com/realsmartmachines/read#leanpub-auto-third-party-projects-based-on-htm)
  + [Appendix 1 - NuPIC Resources](https://leanpub.com/realsmartmachines/read#leanpub-auto-appendix-1---nupic-resources)
  + [Appendix 2: Clortex Resources](https://leanpub.com/realsmartmachines/read#leanpub-auto-appendix-2-clortex-resources)
  + [Appendix 3: Further Reading and Resources](https://leanpub.com/realsmartmachines/read#leanpub-auto-appendix-3-further-reading-and-resources)
    - [HTM Projects and Information](https://leanpub.com/realsmartmachines/read#leanpub-auto-htm-projects-and-information)
    - [Online Courses](https://leanpub.com/realsmartmachines/read#leanpub-auto-online-courses)
  + [Appendix 4: Mathematics of HTM](https://leanpub.com/realsmartmachines/read#leanpub-auto-appendix-4-mathematics-of-htm)
    - [The HTM Model Neuron - Pattern Memory (aka Spatial Pooling)](https://leanpub.com/realsmartmachines/read#leanpub-auto-the-htm-model-neuron---pattern-memory-aka-spatial-pooling)
    - [Transition Memory - Making Predictions](https://leanpub.com/realsmartmachines/read#leanpub-auto-transition-memory---making-predictions)
  + [Draft White Paper 2014](https://leanpub.com/realsmartmachines/read#leanpub-auto-draft-white-paper-2014)
    - [Chapter 1 - Hierarchical Temporal Memory (HTM) Overview](https://leanpub.com/realsmartmachines/read#leanpub-auto-chapter-1---hierarchical-temporal-memory-htm-overview)
    - [Chapter 2: HTM Cortical Learning Algorithms](https://leanpub.com/realsmartmachines/read#leanpub-auto-chapter-2-htm-cortical-learning-algorithms)
    - [Chapter 3: Spatial Pooling Implementation and Pseudocode](https://leanpub.com/realsmartmachines/read#leanpub-auto-chapter-3-spatial-pooling-implementation-and-pseudocode)
    - [Chapter 4: Temporal Pooling Implementation and Pseudocode](https://leanpub.com/realsmartmachines/read#leanpub-auto-chapter-4-temporal-pooling-implementation-and-pseudocode)
    - [Appendix A: A Comparison between Biological Neurons and HTM Cells](https://leanpub.com/realsmartmachines/read#leanpub-auto-appendix-a-a-comparison-between-biological-neurons-and-htm-cells)
    - [Appendix B: A Comparison of Layers in the Neocortex and an HTM Region](https://leanpub.com/realsmartmachines/read#leanpub-auto-appendix-b-a-comparison-of-layers-in-the-neocortex-and-an-htm-region)
    - [Glossary](https://leanpub.com/realsmartmachines/read#leanpub-auto-glossary)
  + [Efficiency of Predicted Sparseness as a Motivating Model for Hierarchical Temporal Memory](https://leanpub.com/realsmartmachines/read#leanpub-auto-efficiency-of-predicted-sparseness-as-a-motivating-model-for-hierarchical-temporal-memory)
    - [Part 1 - Introduction and Description.](https://leanpub.com/realsmartmachines/read#leanpub-auto-part-1---introduction-and-description)
    - [Hierarchy](https://leanpub.com/realsmartmachines/read#leanpub-auto-hierarchy-1)
    - [Regions](https://leanpub.com/realsmartmachines/read#leanpub-auto-regions-1)
    - [Layers](https://leanpub.com/realsmartmachines/read#leanpub-auto-layers-1)
    - [Columns](https://leanpub.com/realsmartmachines/read#leanpub-auto-columns-1)
    - [Neurons](https://leanpub.com/realsmartmachines/read#leanpub-auto-neurons)
    - [Dendrites](https://leanpub.com/realsmartmachines/read#leanpub-auto-dendrites)
    - [Synapses](https://leanpub.com/realsmartmachines/read#leanpub-auto-synapses-3)

**Preface**

**Numenta Platform for Intelligent Computing (NuPIC)**

NuPIC is Numenta’s Open Source Project and is the reference implementation of HTM and CLA. It has a vibrant community of almost 1000 mailing list members, drawn from industry, academia, and interested observers. Matt Taylor is Numenta’s full-time Community Flag Bearer and provides a friendly welcome to all comers.

You can get involved with NuPIC by visiting the [NuPIC Community Website](http://numenta.org/). We have three mailing lists:

1. **[nupic-theory]** for those interested in the HTM and CLA theories and their place in neuroscience. Jeff is a regular contributor and answerer of questions.
2. **[nupic-discuss]** for news and discussion of NuPIC and related subjects.
3. **[nupic-hackers]** for people developing the NuPIC source code itself.

**Clortex**

Clortex is an Open Source Project started by Fergal Byrne. Clortex is based on Jeff Hawkins’ theories and Numenta’s official NuPIC project. I’m hoping to donate or merge Clortex into the NuPIC community as soon as it’s good enough.

You can get involved with Clortex by:

1. Getting the code on the Clortex Github page at <https://github.com/fergalbyrne/clortex>
2. Joining the [Clortex Google Group](https://groups.google.com/d/forum/clortex)
3. Visiting and Liking the [Clortex Facebook Page](https://facebook.com/clortex)
4. Getting updates on [the blog](http://inbits.com/)
5. emailing me at [fergal@brenter.ie](mailto:fergal@brenter.ie)
6. Following me ([@fergbyrne](http://twitter.com/fergbyrne)) on Twitter
7. Sharing a [link to this book on LeanPub.com](https://leanpub.com/realsmartmachines)

In June 2014, I gave a talk on HTM, NuPIC, and Clortex at euroClojure in Krakow, Poland. You can watch it [on Vimeo](http://vimeo.com/100976692).

**Credits**

Thanks to LeanPub.com for creating a great platform for lean publishing.

Thanks also to all the great developers in the Clojure community, who have put together one of the best ecosystems for building sophisticated software and having fun at the same time.

**Introduction**

We’ve dreamed of Machine Intelligence for centuries, and until computers appeared in the 1940’s, it seemed sure to remain in the realm of Science Fiction. For the first few decades of the Computer Age, though, it looked inevitable to computer experts and the public at large that we would soon build intelligent machines. Then, those expectations foundered on our failure to address the unseen complexities of how thinking really works. And now most people again believe that it is, and always will be, just a dream.

This may remind you of the Alchemists’ doomed quests for cheap gold and eternal life.

The human brain is an intelligent machine which we can understand and emulate. While forbiddingly large and complex at first glance, the brain could only function if its components and their operation were at heart quite simple. By finding the basic principles at work in the brain and applying them to build software, people are already creating real machine intelligence and solving real problems for themselves. Hierarchical Temporal Memory, the subject of this book, is both a theory of the brain, and a set of blueprints for machine intelligence.

Based on extraordinary recent advances in the detailed study of information processing in the brain, a simple set of principles has been identified, allowing us to identify and reverse engineer each of the key steps, in enough detail that they can be run on a normal computer.

This book is about a learning machine you can start using today. This is not science fiction, nor some promised technology we’re hoping to see in the near future. It’s already here, ready to download and use. It is already being used commercially to help save energy, predict mechanical breakdowns, and keep computers running on the Internet. It’s at the centre of a vibrant open source community with growing links to leading-edge academic and industrial research. Based on more than a decade of research and development by Jeff Hawkins and his team at Numenta, NuPIC is a system built on the principles of the human brain, a theory called *Hierarchical Temporal Memory* (HTM).

NuPIC stands for *Numenta Platform for Intelligent Computing*. On the face of it, it’s a piece of software you can download for free, do the setup, and start using right away on your own data, to solve your own problems. This book will give you the information you need to do just that. But, as you’ll learn, the software (and its usefulness to you as a product) is only a small part of the story.

NuPIC is, in fact, a working model in software of a developing theory of how the brain works, Hierarchical Temporal Memory. Its design is constrained by what we know of the structure and function of the brain. As with an architect’s miniature model, a spreadsheet in the financial realm, or a CAD system in engineering, we can experiment with and adjust the model in order to gain insights into the system we’re modelling. And, just as with those tools, we can also do useful work, solve real-world problems, and derive value from using them.

And, as with other modelling tools, we can use NuPIC as a touchstone for a growing discussion of the basic theory of what is going on inside the brain. We can compare it with all the facts and understanding from decades of neuroscience research, a body of knowledge which grows daily. We believe that the theories underlying NuPIC are the best candidates for a true understanding of human intelligence and that NuPIC is already providing compelling evidence that these theories are valid.

*Clortex* is a new open source project which complements NuPIC and provides a new architecture for working with HTM in new application areas. Founded by the author, the Clortex project is aimed at bringing HTM to a new audience.

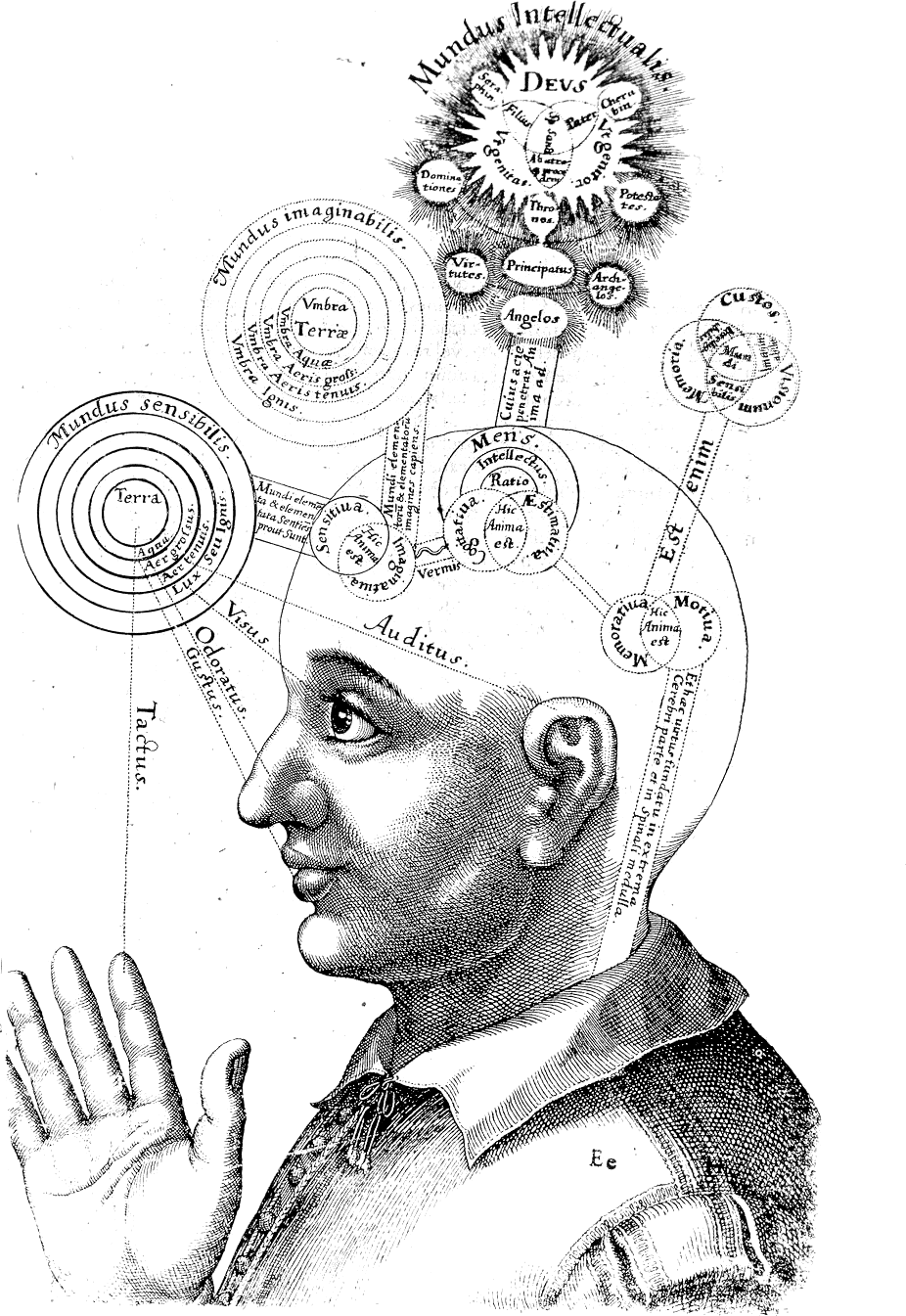
This book begins with an overview of how NuPIC fits in to the worlds of Artificial Intelligence and Neuroscience. We’ll then delve a little deeper into the theory of the brain which underlies the project, including the key principles which we believe are both necessary and sufficient for intelligence.

In Chapter 3, we’ll see how the design of NuPIC corresponds to these principles, and how it works in detail. Chapter 4 describes the NuPIC software at time of writing, as well as its commercial big brother, Grok. Finally, we’ll describe what the near future holds for HTM, NuPIC, and Grok, and how you can get involved in this exciting work. The details of how to download and operate NuPIC are found in the Appendices\*, along with details of how to join the NuPIC mailing list.

**Pending completion of the Appendices, please see the Preface for details on NuPIC and Clortex**.

**One: Perspective**

**Brain Theory as a Basis for AI**



This book is about a new theory of how the brain works, and software which uses this theory to solve real-world problems intelligently in the same way that the brain does. In order to understand both the theory and the software, a little context is useful. That’s the purpose of this chapter.

Before we start, it’s important to scotch a couple of myths which surround both Artificial Intelligence (AI) and Neuroscience.

The first myth is that AI scientists are gradually working towards a future human-style intelligence. Despite what they tell us, and they themselves believe, they are really building computer programs which merely appear to behave in a way which we might consider smart or intelligent - as long as we ignore how they work. Don’t get me wrong, these programs are very important in our understanding of what constitutes intelligence, and they also provide us with huge improvements in understanding the nature and structure of problems solved by brains. The difficulty is that brains simply don’t work the way computer programs do, and there is no reason to believe that human-style intelligence can be approached just by adding more and more complex computer programs.

The other myth is that Neuroscience has figured out how our brains work. Neuroscience has collected an enormous amount of data about the brain, and there is good understanding of some detailed mechanisms here and there. We know (largely) how individual cells in the brain work. We know that certain regions of the brain are responsible for certain functions, for example, because people with damage there exhibit reduced efficiency in particular tasks. And we know to some extent how many of the pieces of the brain are connected together, either by observing damaged brains or by using modern brain-mapping technologies. But there is no systematic understanding which could be called a Theory of Neuroscience, one which explains the working of the brain in detail.

**Traditional Artificial Intelligence**

Traditional AI does not provide a basis for human-like intelligence. In order to understand the reasons for this, let’s take a look inside a digital computer.

A computer chip contains a few billion very simple components called transistors. In digital circuits, transistors act as a kind of switch (or relay): they allow a signal through or not, based on a control signal. Computer chip, or hardware, designers produce detailed plans for how to combine all these switches to produce the computer you’re reading this on. Some of these transistors are used to produce the logic in the computer, making decisions and performing calculations according to a program written by others: software engineers. The program, along with the data the program uses, are stored in yet more chips – the memory – using transistors which are either on or off. The on or off state of these memory *bits* comprise a code which stands for data – whether numbers, text, image pixels, or program codes which instruct the computer what instruction to perform at a particular time.

If you open up a computer, you can clearly see the different parts. There’s a big chip, usually with a fan on top to cool it, called the Central Processing Unit or CPU, which is where the hardware logic is housed. Separate from this, a bank of many smaller chips houses the Random Access Memory (RAM) which is the fastest kind of memory storage. There will also be either a hard disk or a solid state disk, which is where all your bulk data - programs, documents, photos, music and video - are stored for use by the computer. When your computer is running, the CPU is constantly fetching data from the memory and disks, doing some work on it, and writing the results back out to storage.

Computers have clearly changed the world. With these magical devices, we can calculate in one second with a spreadsheet program what would have taken months or years to do by hand. We can fly unflyable aircraft. We can predict the weather ten days ahead. We can create 3D movies in high definition. We can, using other electronic “senses”, observe the oxygen and sugar consumption inside our own brains, and create a map of what’s happening when we think.

We write programs for these computers which are so well thought out that they appear to be “smart” in some way. They look like they’re able to out-think us; they look like they can be faster on the draw. But it turns out that they’re only good at certain things, and they can only really beat us at those things. Sure, they can calculate how to fly through the air and get through anti-aircraft artillery defences, or they can react to other computer programs on the stock exchange. They seem to be superhuman in some way, yet the truth is that there is no skill involved, no knowledge or understanding of what they’re doing. Computer programs don’t learn to do these amazing things, and we don’t teach them. We must provide exhaustive lists of absolutely precise instructions, detailing exactly what to do at any moment. The programs may appear to behave intelligently, but internally they are blindly following the scripts we have written for them.

The brain, on the other hand, cannot be programmed, and yet we learn a million things and acquire thousands of skills during our lives. We must be doing it some other way. The key to figuring this out is to look in some detail at how the brain is put together and how this structure creates intelligence. And just like we’ve done with a computer, we will examine how information is represented and processed by the structures in the brain. This examination is the subject of Chapter Two. Meanwhile, let’s have a quick look at some of the efforts people have made to create an “artificial brain” over the past few decades.

Artificial Intelligence is a term which was coined in the early 1950′s, but people have been thinking about building intelligent machines for over two thousand years. This remained in the realm of fantasy and science fiction until the dawn of the computer age, when machines suddenly became available which could provide the computational power needed to build a truly intelligent machine. It is fitting that some of the main ideas about AI came from the same legendary intellects behind the invention of digital computers themselves: Alan Turing and John von Neumann.

Turing, who famously helped to break the Nazi Enigma codes during WWII, theorised about how a machine could be considered intelligent. As a thought experiment, he suggested a test involving a human investigator who is communicating by text with an unknown entity – either another human or a computer running an AI program. If the investigator is unable to tell whether he is talking to a human or not, then Turing considers the computer to have passed his test and must be regarded as “intelligent” by this definition. This became known as the Turing Test and has unfortunately become a kind of Holy Grail for AI researchers for more than sixty years.

Meanwhile, the burgeoning field of AI attracted some very smart people, who all dreamed they would soon be the designer of a machine one could talk to and which could help one solve real-world problems. All sorts of possibilities seemed within easy reach, and so the researchers often made grand claims about what was “just around the corner” for their projects. For instance, one of the milestones would be a computer which could beat the World Chess Champion, a goal which was promised within 5 years, every year since the mid-50s, and which was only achieved in the 21st century using a huge computer and a mixture of “intelligent” and “brute-force” techniques, none of which resembled how Gary Kasparov’s brain worked.

Everyone recognised early on that intelligence at the level of the Turing Test would have to wait, so they began by trying to break things down into simpler, more achievable tasks. Having no clue about how our brains and minds worked as machines, they decided instead to theorise about how to perform some of the tasks which we can perform. Some of the early products included programs which could play Noughts and Crosses (tic-tac-toe) and Draughts (checkers), programs which could “reason” about placing blocks on top of other blocks, in a so-called micro-world, and a program called Eliza which used clever and entertaining tricks to mimic a psychiatrist interviewing a patient.

Working on these problems, developing all these programs, and thinking about intelligence in general has had profound effects beyond Computer Science in the last sixty years. Our understanding of the mind as a kind of computer or information processor is directly based on the knowledge and understanding gained from AI research. We have AI to thank for Noam Chomsky’s foundational Universal Grammar, and the field of Computational Linguistics is now required for anyone wishing to understand linguistics and human language in general. Brain surgeons use the computational model of the brain to identify and assess birth defects, the effects of disease and brain injuries, all in terms of the functional modules which might be affected. Cognitive psychology is now one of the basic ways to understand the way that our perceptions and internal processes operate. And the list goes on. Many, many fields have benefited indirectly from the intense work of AI researchers since 1950.

However, traditional AI has failed to live up to even its own expectations. At every turn, it seems that the “last 10%” of the problem is bigger than the first 90%. A lot of AI systems require vast amounts of programmer intelligence and do not genuinely embody any real intelligence themselves. Many such systems are incapable of flexibly responding to new contexts or situations, and they do not learn of their own accord. When they fail, they do not do so in a graceful way like we do, because they are brittle and capable only of working while “on-tracks” in some way. In short, they are nothing like us.

Yet AI researchers kept on going, hoping that some new program or some new technique would crack the code of intelligent machine design. They have built ever-more-complex systems, accumulated enormous databases of information, and employed some of the most powerful hardware available. The recent triumphs of Deep Blue (beating Kasparov at chess) and Watson (winning at the Jeopardy quiz game) have been the result of combining huge, ultra-fast computers with enormous databases and vast, complex, intricate programs costing tens of millions of dollars. While impressive, neither of these systems can do anything else which could be considered intelligent without reinvesting similar resources in the development of those new programs.

It seems to many that this is leading us away from true machine intelligence, not towards it. Human brains are not running huge, brittle programs, nor consulting vast databases of tabulated information. Our brains are just like those of a mouse, and it seems that we differ from mice only in the size and number of pieces (or regions) of brain tissue, and not in any fundamental way.

It appears very likely that intelligence is produced in the brain by the clever arrangement of brain regions, which appear to organise themselves and learn how to operate intelligently. This can be proven in animal research labs, when experimenters cut connections, shut down some regions, breed mutants, and so on. There is very little argument in neuroscience that this is how things work. The question then is: how do these regions work in detail? What are they doing with the information they are processing? How do they work together? If we can answer these questions, it is possible that we can learn how our brains work and how to build truly intelligent machines.

I believe we can now answer these questions. That’s what this book claims to be about, after all!

**Machine Intelligence versus Machine Learning**

There is a branch of Computer Science and AI which has recently gained, or regained, prominence, partly as a result of the current age of Big Data: Machine Learning. Machine Learning involves having computer systems learn to carry out a task or solve a problem automatically, without having to explicitly instruct them about every step.

Two main branches appear to dominate the field of Machine Learning. The first bears some similarity to traditional AI in that it uses mathematical, statistical, and symbolic programming to compute its results. Essentially, the software searches a space of possibilities to identify the best function to use to solve the given problem; this kind of approach is used, for example, to predict trends in economic data into the future, to guess who’s going to win an election, or to identify the probability that a Large Hadron Collider experiment has found the Higgs Boson.

The other branch is based on some kind of “neural net”, a network of simplified artificial processing elements which combine to learn or model some aspects of the data in order to make predictions or identify patterns. Invented at the very beginning of the computer age, these networks have undergone many phases of mania and depression, as fundamental limitations were first ignored, then admitted, and finally overcome as the techniques improved. Recently, nets known as Deep Learning nets and Convolutional Neural Networks (ConvNets) have become very popular, being used as the core of search engines, speech recognisers, and other applications by giants like Google and Facebook, and their inventors now sit at the top tables of global corporations as “VP of AI Research” and “Chief Scientist”.

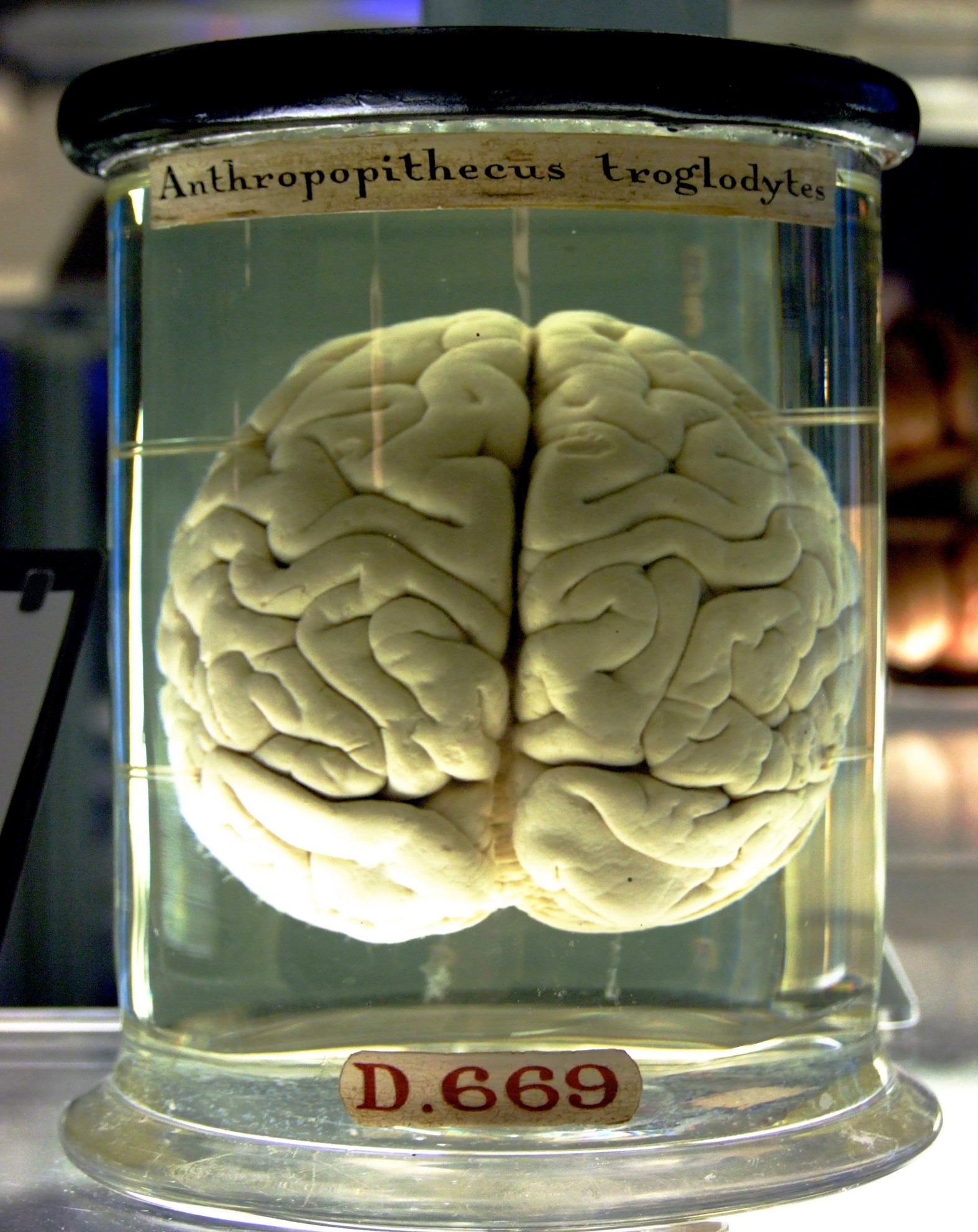
Neural nets attempt to model how the brain processes information. They work by connecting up pairs of neurons with adjustable “weights”, which indicate the strength of the influence of one neuron on another. The neurons are arranged in layers (as in the brain), input data is fed in at the bottom, passed up through the layers until it comes out, transformed by all the weights and summations, at the top, where it can be used to make predictions or identify what the network is looking at. Classic tasks for neural networks include identifying whether a picture is of a cat or a dog, deciphering handwritten characters, interpreting speech from sound, and so on. At tasks like this, the current world champions are all some kind of Deep Learning system or ConvNet.

While very interesting and quite well understood theoretically, we believe that such simple modelling of the details of brain function is yet another case of a fundamentally limited technology which will eventually fail to realise high expectations. One of the reasons for this is that neural net researchers seem to insist that their systems (and the way those systems settle on a solution) can be described using very simple mathematics, and that certain properties of the systems can be proved mathematically. They dare not make their models any more faithful to the true structure of the brain (as we do), because the mathematics quickly becomes too difficult to support such proof-based reasoning about them.

I would, however, encourage the reader to learn all about these systems. In certain ways they’ve gone much further than HTM researchers in addressing many cognitive tasks - they’ve been at this for decades, and they’ve had support from governments and, more recently, the likes of Google and Facebook, so there is already a large community of extraordinarily smart people hard at work to test the limits of what can be achieved. Many discoveries have emerged from their research, including many on perception, pattern recognition, hierarchy, and representation. Links to books, articles, courses and talks on neural nets can be found in the Further Reading section, including a page explaining HTM to experts in Machine Learning.

**Two: Jeff Hawkins’ Theory of the Neocortex**

**Hierarchical Temporal Memory and the Cortical Learning Algorithm**



A chimpanzee brain preserved in a jar

**A Little Neuroscience Primer**

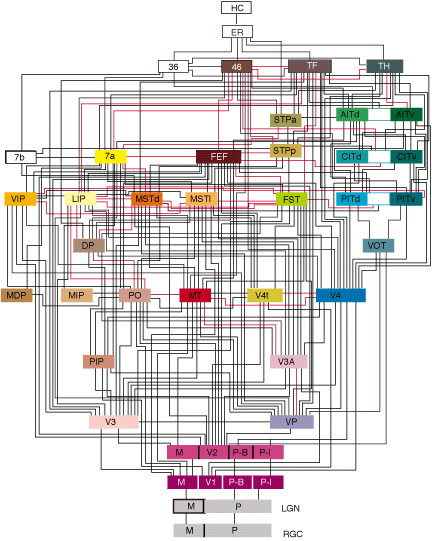
Brain, n.: an apparatus with which we think we think. > *Ambrose Bierce, The Devil’s Dictionary*

The seat of intelligence is the neocortex (literally “new bark” in Latin) which is the crinkly surface of your brain. This latest addition to the animal brain is only found in mammals, and we have the biggest one of all. The neocortex is crumpled up so that it fits inside the skull; in fact it’s a large surface about the size of a dinner napkin, and has a thickness of about 2mm. The neocortex contains about 60 billion neurons (“brain cells” or “grey matter” - the surface of the brain) which form trillions of connections among themselves (“white matter”, much of which is underneath the crumpled surface) in such a way as to create what we know as intelligence.

Clearly, the neocortex is some kind of computer, in the sense that it is an organ for handling information. But its structure is completely unlike a computer – it has no separate pieces for logic, memory, and input-output. Instead, it is arranged in a hierarchy of patches, or regions, like the staffs and units in an army. At the bottom of the hierarchy, regions of neurons are connected to the world outside the brain via millions of sensory and motor nerves.

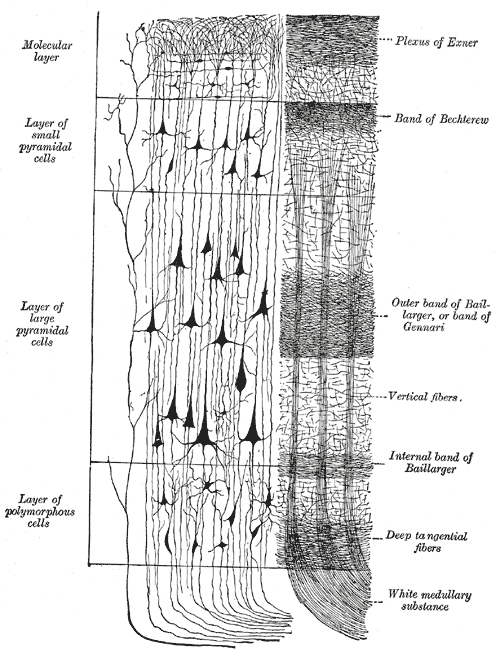
Each region takes information from below it, does something with that information, and passes the results up the chain of command. Each region also takes orders of some sort from higher-ups, does something with those orders and passes on, or distributes, more detailed orders to regions below it. Somewhere during this process, the brain forms memories which are related to all this information passing up and down.

The following diagram shows the hierarchy of the macaque monkey’s brain, the part which processes visual information from the eyes. Note that each line in the diagram is more like a big cable, containing hundreds, thousands, or even millions of individual connections between regions.



Visual hierarchy of the macaque

Within this hierarchy, each region is remarkably similar in structure. Every region seems to be doing the same thing in the same way as all the others. If you look closely at practically any small piece of neocortex, you’ll see the same pattern again and again, as shown below.



Drawing showing layer structure

As you see in this drawing from Gray’s Anatomy (1918!), neurons are organised vertically in columns, and each column appears to be divided into a number of layers (usually 5 or 6), each of which contains its own “type” of neuron. In addition to what you see here, there are huge numbers of horizontal connections between neighbouring columns; these connections, in fact, add up to almost 90% of all connections in the neocortex.

Two important facts may be observed at this level of detail. As suggested by the diagram above, each column shares inputs coming in from below, so the cells in a column all respond similarly to the same inputs. Secondly, at any time only a few percent of columns in a region will be active, the others being largely quiet.

**Neuroscience in Plain Language**

Everyone knows that brains and neurons are tremendously complicated things, perhaps so complex that it’s impossible to understand how they work in any real detail. But a lot of the complexity is only because neurons and brains are living things, which evolved over tens of millions of years into what we see today. Another source of complexity for many people is the use by scientists of words and language which adds little to understanding and tends to confuse or intimidate.

In this section, I’m going to strip away as much as I can of the “bioengineering” (things needed for real cells to keep working), and scientific language, which seems to prefer long words - and compounds of them - to short, memorable ones, which might do a better job. For those who prefer the full neuroscience treatment, I tell precisely the same story in the form of a scientific paper which you’ll find in the Appendix (neuroscientists might find it instructive to compare the two and reflect on their own practises..).

This approach has been inspired by ideas from Douglas Höfstadter, Stephen Pinker, Francis-Noël Thomas & Mark Turner, and the wonderful “Edge of the Sky” by Roberto Trotta.

**Two Kinds of Neurons**

There are many, many kinds of neurons in the brain, each one “designed” by evolution to play a particular role in processing information. We still haven’t figured out the complete zoology, never mind assigned job titles to each variety. Of course, Nature may have a purpose for all this diversity, but it also might just have happened to have worked out this way, and in any case a lot of it might be there because of the demands and limits of keeping billions of cells fed and watered over a lifetime.

As you’ll see, we’ll be able to get a perfectly good understanding of brain function, by pretending there are just two kinds of neurons: those that *excite* others (or make them fire), and those that *inhibit* others (or stop them from firing). Since these names already have many syllables, and will force me to use (and type out) tongue-twisting words like *excitatory* and *inhibitory*, I’m going to use the names **spark**and **snuff** for these cells (handily, I can use these as verbs as well).

Why do we need two types of cell? Well, spark cells are needed to transmit signals (sparks) from one place to another, to gather up sparks from various sources and make the decision about whether to send on a new spark. But if all we had were spark cells, every incoming signal (say from your ears) would result in a huge chain reaction of sparking cells all over the brain, because each spark cell is connected to thousands of others. The snuff cells are there to control and direct the way the spark cells fire, and they are in fact critical to the whole process.

**How Neurons Work**

Both types of cell work in almost the same way, the only difference being what kinds of effect they have on other cells. They both have lots of **branches** (called *dendrites* after the Latin for “tree”), which gather sparks of signal from nearby cells. In the cell’s **body**, these sparks “charge up” the cell like a really fast-charging battery (for electronics nerds, it’s really more like a capacitor). This is very like the way a camera flash works - you can often hear it “charge up” until a light signals it’s ready to “fire”. If there are enough sparks, the cell charges up until it’s fully charged, and it then fires a spark of its own down a special output fibre called its **axon** (which just means “axis” in Greek).

Because the cell is just a bag of molecules, and also due to some machinery inside the cell, charge tends to leak out of the cell over time (this happens with your camera’s flash too, if you don’t take a picture soon after charging it up). The branches gathering sparks are even more leaky, which we’ll see is also important. In both cases, the cell needs to use all the sparks it receives as soon as it can. If it doesn’t fire quickly, all the charge leaks away and the cell has to start again from scratch (known as its *resting potential*).

The spark travels down the axon, which may split several times, leading to a whole set of sparks travelling out to many hundreds or thousands of other cells. Now, it’s when a spark reaches the end of the axon that we see the difference between the two kinds of cell. The axon carrying the spark from the first cell almost meets a branch of the receiving cell, with a tiny **gap** (called a *synapse*) between the two. This gap is quite like the spark gap in your car’s engine, only the “sparks” are made of special chemicals called *neurotransmitters*.

There are two main kinds of neurotransmitters, one for sparking cells and the other for snuffing. Each type of cell sends a different chemical across the gap to the receiving cell, which is covered in socket-like *receptors*, each tuned to fit a particular neurotransmitter and cause a different reaction - either to assist the cell in charging up and making its own spark, or else to snuff out this process and shut the cell down.

There’s one more important twist, and it’s in the way the incoming sparks are gathered by the cell’s branches. On **near branches**(*proximal dendrites*), the branches are so thick and close to the body that they act like part of it, so any sparks gathered here go straight in and help charge up the cell. Further out, on **far branches** (*distal dendrites*), the distance to the body is so great, and the branches so thin and leaky, that single sparks will fizzle out long before they have any chance to charge up the cell.

So, Nature has invented a trick which gathers a *set* of incoming sparks - all of which have to come in at nearly the same time, and into a short **segment** of the branch - and generates a **branch-spark** (*dendritic spike*), which is big enough to reach all the way to the cell body and help charge it up.

Thus we see that each neuron uses incoming signals in two different ways: directly on near branches, and indirectly, when many signals coincidentally appear on a short segment of a far branch. We’ll soon see that this *integration* of signals is key to how the neocortex works.

If you’re familiar with “traditional” neural networks (*NNs*), you might be scratching your head at the relative complexity of the neurons I’ve just described. That’s understandable, but you’ll soon see that these extra features, when combined with some structures found in the brain (which we’ll get to), are sufficient - and necessary - to give networks of HTM neurons the power to learn patterns and sequences which no simple NN design can. I’ve included a section in the Appendix which explains how you can think of each HTM neuron as a kind of NN in its own right.

**The Race to Recognise - Bingo in the Brain**

When someone of my vintage hears the word “Bingo”, I picture a run-down village hall full of elderly ladies whiling away their days knitting, gossiping and listening out for their numbers to be called out. The winner in Bingo is the first lady to match up all the numbers on her particular card and shout out the word “House!” or “Bingo!” Whoever does this first wins the prize, leaving everyone else empty-handed - even those poor souls just one or two numbers short. The consolation is that losers this time might hit the jackpot combination in the very next round.

The same game is played everywhere in the brain, but the players are spark cells, the cards the cells’ *near branches*. The numbers on each cell’s cards are the set of cells it receives sparking signals from. Each “called number” corresponds to one of these upstream cells firing, and so every cell is in a race to hear its particular combination of numbers adding up to Bingo!

**Inhibition**

The winner-takes-all feature of the game is mirrored using the snuff cells, a process known as *inhibition*. Mixed among the sparking cells, snuff cells are waiting for someone to call out Bingo! so that they can “snuff out” less successful cells nearby. The snuff cells are also connected one to another, but in this case in a sparking sense, so the snuffing effect spreads out from the initial winner until it hits cells which have already been snuffed out.

How might these other cells already been snuffed out? Well, it’s as if the Bingo hall is so big, with so many players, that there are many games going on at the same time, with Bingo callers scattered at intervals around the hall. Each old lady is within earshot of at least one, but possibly several, Bingo callers, and they might be able to combine numbers from any or all of them. As all these games proceed in parallel, we are likely to see calls of Bingo! arising all over the hall at any time, sometimes almost simultaneously. There’ll usually be some distance between any two winning ladies, since the chances of them matching the same numbers from the same callers with their different cards are low.

When someone right next to you stands up and shouts Bingo!, you must tear up your card and start afresh, since this round of the game is now over. Similarly, if someone beside you (or in front of or behind you) tears up her card, that’s treated as a signal to tear up your own card too.

Thus we would observe a pattern of ladies standing up - seemingly at random - all over the hall, followed by a wave of card-tearing which spreads out from each winner, ending only when it meets another wave coming the other way from a different winner (you can’t tear up a card you’ve already torn up).

This kind of pattern, where a small number of ladies, each standing alone in a sea of disappointed runners-up, is called a *sparse pattern*, and it’s an important part of how the neocortex works.

**Recognition**

The next important feature of this neural Bingo! game is that every lady, whether they won or lost the last time, gets the *same card* every round (almost: they’ll get to “cheat” soon, you’ll see). This means that each lady is destined to win if their usual set of numbers comes up fast enough to beat their near neighbours.

To make it a bit less boring for the ladies, we’ll relax the usual rules of Bingo! to allow a lady to win the game if she gets over a certain *threshold* of called-out numbers, rather than having to get a full house. This greatly increases the chances of winning, because you can miss out on several numbers and still get enough to pass the threshold. Each lady is now capable of responding to a set of combinations, or *patterns* of numbers, and she’ll win out whenever one of these lists of numbers actually gets called out (mixed in with several numbers of no interest to this particular lady).

We can see now that the ladies are competing to *recognise* the pattern of numbers being produced by the Bingo callers, and that they will form a pattern which in a sense *represents* that input pattern. If the callers produce a different set of numbers, we will have a different pattern of ladies standing up, and thus a different representation. Further, swapping out one or two numbers in each caller’s sequence will only have a small effect on the pattern, since each lady is using many numbers to match her card, and she’s still likely to win out over nearby ladies who have only half their cards filled out.

This kind of pattern, which has few active members and mostly inactive ones, and in which the choice of winners represents the set of inputs in a way which is robust to small changes in detail, is called a **Sparse Distributed Representation** or **SDR**, and it’s one of the most important things in HTM theory. We’ll see later how and why SDRs are used all over the place, in both neocortex and the software we’re building to use its principles.

**Memory and Learning**

One last thing: the ladies, being elderly, are a bit hard of hearing. This means that they’ll occasionally mishear a number being called out, and fail to recognise that they’ve made another bit of progress towards being this round’s winner. This is in fact a real issue in the brain, where all kinds of biological factors are likely to interfere with the transmission of all the tiny sparks of electrical and chemical signals. There’s therefore always a fairly significant chance that each number called will never reach a lady’s card when it should have.

On the other hand, since the ladies always get the same cards each round, they’ll get used to listening out for their particular set of numbers, especially when it leads to them winning the game. Numbers which seldom get called out, or rarely contribute to them shouting Bingo! will, conversely, be somewhat neglected and even ignored by a particular lady. Each lady thus learns to better recognise her particular best numbers and this makes it more likely that she’ll respond with a shout when this pattern recurs.

In the brain, the “hearing” or failure to hear occurs in the gaps between neurons, in the synapses. The gaps are actually between a bulb on the axon of the sending (*presynaptic*) cell and a **spine** on the branch of the receiving (*postsynaptic*) cell. The length, thickness and number of receptors on the dendritic spine changes quite quickly (in the order of seconds to minutes) in response to how much the signals crossing correspond to the firing of the receiving cell.

When the signals match the activity of the cell, the spine grows thicker and longer, closing the gap and improving the likelihood that the receiving cell will “hear” the incoming signal. In contrast, if the signals mismatch with the cell’s activity, the spine will tend to dwindle and the gap will widen. Transmitting irrelevant signals is wasteful of resources, so the cell is built to allocate more to its best-performing synapses (and recycle material from the poor performers).

A spine which grows close enough to shrink the gap to a minimum is practically guaranteed to reliably receive any incoming spark from the axon producing it, while one which is short and thin will almost certainly not “hear” its number being called. In between, there’s a grey area which corresponds to a probability of the signal being transmitted. There’s a “degree of connectedness” corresponding to each synapse, varying from “disconnected” to “partially connected” to “fully connected”.

This strengthening and weakening of synaptic connections is precisely how learning (and forgetting) occurs in the brain. **Memory** is contained in the vast combination of synapses connecting the billions of neurons in the neocortex.

**Meaning**

So, what do these numbers, and patterns of numbers, *mean*? Well, think of the popular drinking games played all over the world by young people, variations on the Bingo! game such as politician-bingo, where each player has a list of hackneyed words or phrases used by the most vapid politicians (“going forward”, “people power”, “hope” and “responsibility”). In this case the numbers are replaced by meaningful symbols, and each elderly lady (or hipster in this case) is competing to recognise a certain pattern of those symbols.

This is precisely how the brain works. Initially, signals are coming in from your eyes, ears and body, and the first region of neurons each set of signals reaches will have a huge game of sensory Bingo! and form a pattern which represents the stuff you’re seeing, hearing or sensing. This pattern reflects tiny details such as a short edge in a particular spot in your visual field, and it gets passed up to higher regions which (again using Bingo!) put together patterns of these features, perhaps representing a surface composed of short edges. This process repeats all the way up to Bingo! games which recognise and represent big objects such as words, dogs, or Bill Clinton.

Note that the ladies are just checking numbers against their cards; they don’t *themselves* have any understanding of the meanings. In the same way, each neuron just receives anonymous signals, fires if it gets to, and does some reinforcement on its synapses.

The meaning of a neuron firing is *implicit* in the pattern of connections it receives signals from. (These might turn out to correspond to a cross-shaped or diamond-shaped feature in vision, for example, because that particular neuron represents a combination of sensory neurons which each fire in the presence of a short line segment forming part of that pattern.) The meaning a neuron represents is also *learned*, since it reinforces its connections with good providers of signals causing it to fire, so it learns to best represent those patterns which statistically lead to its successful firing.

In the bingo hall, each lady can hear some nearby numbers being called, but many callers are too far away for all of them to be heard. Thus, as well as having their own set of numbers on the card, each lady hears only a particular subset of the full set of numbers being called out across the hall. Somewhere in another corner, we could have a lady with a very similar card, who is hearing a somewhat different set of numbers being called out. In many cases, both these ladies are likely to win if one of them does, and to lose out to a better pair of ladies when their numbers don’t appear.

There may be many such sisters-in-arms who are all likely to win together. Thus, we can see that the representation is **distributed** across the hall, with a set of ladies responding together to similar sets of numbers. This representation is robust to the kind of disruption caused by faulty hearing, ladies occasionally falling asleep, or (sadly, inevitable among thousands of elderly ladies), the passing on of the odd player.

**Capacity I**

Thus far, we’ve described a neural pattern recognition system, where we have many neurons doing nothing, and a small number firing at any time, forming a sparse pattern which represents the recognised input. Since we’re only using a tiny number of neurons at a time, surely we’re wasting a lot of capacity to store a representation of our input patterns? Well, a system like this is much less space-efficient than a typical “dense” representation as used in computers and their files, but it turns out that even a modestly large SDR can represent a surprising number of patterns.

For example, we typically use an SDR of 2048 cells with 2%, or 40 cells active for each pattern. This kind of SDR can represent 1084different patterns, which is written as 1,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000! This is more than a billion times the number of atoms in the known universe.

For a detailed numerical treatment of the properties of SDRs, see this recent paper by Subutai Ahmad.

**Prediction**

Now, our Bingo ladies, being active and somewhat trendy, are all on Facebook, and they enjoy crowing about their skill at Bingo among their friends. Whenever a lady wins, stands up and shouts “Bingo!”, she gets straight on her mobile and posts about it to all her thousands of Facebook friends (who also happen to be somewhere in the hall).

Now, because the numbers have meaning, reflecting some truth about the outside world, there is always some structure to the patterns of numbers as they’re called out around the hall. If an object has a short vertical edge in one place, it’ll have an edge above and below that point (each short edge forming a longer line). So the corresponding set of winning ladies will tend to form groups who win together (at least give or take the odd lady) or fail together. So the Facebook posts will tend to come in groups too. Canny ladies, who remember their turn to win came shortly after they received a bunch of texts in the past, will learn to predict that they’re about to strike lucky.

In the neocortex, the “Facebook messages” are what the far branches are listening for. Each segment corresponds to a group of previous winning cells, which all fired around the same time in the past. When a group of “friends” win, they’ll all send signals to a particular segment, which will cause a branch-spark to pass down and into the cell body. This will partially charge up the cell body, giving it a shortcut to reaching “fully charged” when combined with the sparks coming in from the near branches. We call a cell which is partially charged (*depolarised*) due to activity on far branches, **predictive**.

Like the near branches of a cell, these distal segments will learn to associate a group of incoming signals from the recent past with their own activity, using exactly the same kind of spine growth to reinforce the association. In this way, each cell learns to combine its current “spatial” input on near branches, with the predictive assistance provided by recent patterns of activity.

**Columns of Neurons**

We’ll leave the Bingo hall behind for now (I probably stretched that analogy far enough!), and return to neurons alone. So far, we’ve imagined a large flat surface covered in neurons, all in a race to fire and snuff out their neighbours. Already, this kind of system can perform a very sophisticated kind of pattern recognition, and form a Sparse Distributed Representation of its inputs in the face of significant noise and missing data.

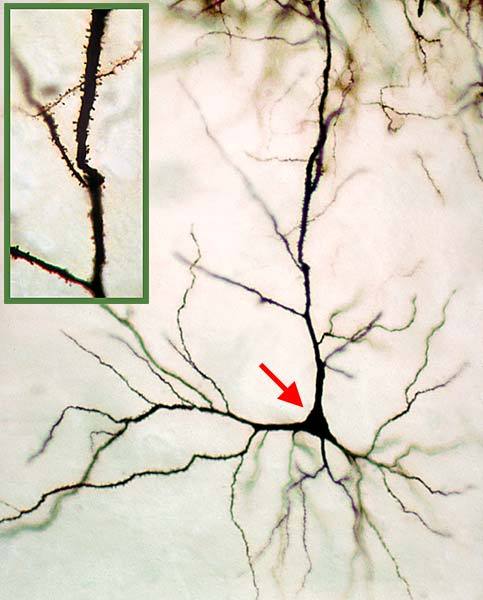
If we simply add in prediction as introduced in the last section, we see that the system has gained new power. Firstly, given the context of previous activity, the layer of neurons can use predictive inputs to “fill in” gaps in its sensory inputs and compensate for even more missing data or noise. Secondly, at any moment the layer is using the current pattern to predict a set of future inputs, which is the first step in learning sequences.

A layer composed of singe cells is limited to predicting one step ahead. To see why, note that at any time, there is information about the current input, represented by the pattern of firing cells, and also information about what the layer predicts will come next, represented by the pattern of predictive depolarisation in cells connected to the current pattern. Conversely, the predictive pattern is based on information only one step in the past, and the layer can’t learn to distinguish between different sequences where the difference is two or more steps in the past. Thus it “forgets” any context which is more than one time step old.

Clearly, our brains have the facility to learn much longer sequences than this. Most of our powerful abilities come from combining high-order sequences into single “chunks” and thus very efficiently storing rich sensory memories and behavioural “scripts”. So we need to extend our model of a layer to fulfil this important requirement.

The trick is to replace each cell with a column of cells, and to use this new structure in a particular way to produce exactly the kind of high-order sequence learning we seek. In addition to having many cells per column, we’ll change the way the snuff cells are organised, and you’ll see that these two changes give rise to a simple, yet very powerful system with exactly the properties we need (and a few more which we’ll want later!).

Zoom in even closer, and you’ll see the structure of an individual neuron.



The “pyramidal” part is the cell body, marked with an arrow. The branching lines surrounding the cell body form a tree (or arbor) of dendrites, each of which is covered in tiny dendritic spines (the little barbs visible in the inset picture). Each spine represents a connection, called a synapse, with another neuron from which it can receive signals. A typical cell in the neocortex potentially receives input from several thousand others.

There are two types of dendrite in a pyramidal cell. The first kind, *proximal* (or near) dendrites, are situated near the cell body and receive almost all their input from “below” in the hierarchy (or from the senses) – this is called feedforward input. The proximal dendrites appear to “add up” the activation signals coming in from below, and will cause the cell to become more and more active depending on the number of incoming signals. When sufficient inputs arrive on the proximal dendrites, the cell may pass a threshold and become active. It will fire off electrical spikes along its axon (or output connection), and this signal will reach all the cells (in this region and others) which receive inputs from the cell. A typical cell might have dozens to a few hundred feedforward connections. Proximal dendrites are responsible for receiving information from the senses (or from other parts of the brain).

The second kind, which accounts for the very large branching seen in the image, are the *distal* (far) dendrites. These extend quite a distance from the cell body and mainly form connections with nearby cells in the same layer. Distal dendrites are also different in how they respond to their inputs. Unlike the proximal dendrites, which respond in a gradual, linear fashion to a number of inputs across them all, distal dendrites will respond only when a number of signals appear close together along a dendrite segment, and also close together in time. If sufficient coincident inputs are received on a dendrite segment, the cell may become *predictive*, as if anticipating that it will become active. Distal dendrites are responsible for the information processing within each region.

This is only a tiny subset of all the knowledge we have about the brain, but it’s the core of what we’ll discuss in this book. Let’s move now to the overarching ideas of Jeff Hawkins’ theory – the Memory Prediction Framework.

**The Memory Prediction Framework**

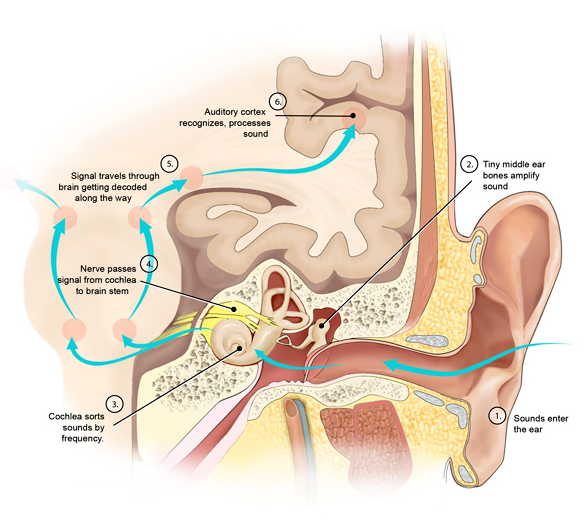
Jeff Hawkins, the author of the theory, tells the story of how he emerged from Cornell as an engineer, and set out to study the brain and figure out how it worked. He approached the people at the famed MIT Artificial Intelligence Lab, looking for an opportunity to research the brain in order to figure out how it works, and thereby help in the development of computer-based intelligence. He was abruptly told that this was not how it was done, that nobody needed to study how the brain does intelligence in order to build AI programs. The converse happened when he approached the subject in the opposite direction, seeking a research post which would use computational ideas to formulate a working theory of the brain. So, as he relates in his book, *On Intelligence*, he entered the technology business for a “few years” so that he could fund his own research into a computational theory of the brain.

Fifteen years later, in 2002, having created the PalmPilot and the Handspring Treo, Hawkins set up the Redwood Center for Neurological Research, wrote *On Intelligence* a year later, and, in 2005, formed Numenta to build a commercial product based on his theory of the brain. This product, Grok, its open-source core, NuPIC, and the underlying theory they’re based on, are the subjects of this book.

The key idea of Hawkins’ initial theory is that the brain is a **memory-prediction machine**. It receives millions of inputs from the body and eyes - from the world - every second, and uses this information to learn (i.e., create, refine) models of the world which it can then use to make predictions and take actions. This is happening at all scales in the brain, which uses its hierarchy to embody the hierarchical structure of the world (including the structure of our own minds).

In his book, Hawkins outlines his central belief that intelligence is the presence of an internal mechanism which learns, based on sequences of perceptions (patterns), to form a model of the world, uses this to predict future events, and, in turn, may generate new pattern sequences which can often be observed as thought or behaviour. Thought, behaviour, and memory are all equivalent to stored or generated sequences of patterns in the brain. Hawkins’ theory – Hierarchical Temporal Memory – is a model of how the brain literally builds itself out of the never-ending procession of information coming in from the senses.

Let’s look at an example which will illustrate how this works. When you’re listening to someone speaking, your ears receive a very fast stream of sounds which are analysed by an organ called the cochlea in your inner ear. The cochlea, which resembles a snail shell, is an ever-narrowing tube of jelly wound up in a spiral. Embedded in the jelly are tiny hairs which have nerve endings (or receptors) at their bases. As the cochlear tube gets narrower along its length, the passing sound waves will cause the jelly to wobble in sympathy, or resonate, when the width matches the pitch of the sound. This means that each hair, and thus each receptor, will fire in response to a certain pitch (or frequency) range in the sound.



We have about 30,000 separate receptors in our ears, so as you listen to someone talking, your brain is receiving a fast-changing stream of signals along 30,000 separate channels at the same time. Most of these signals will be off, with perhaps only 5-10% of them on. The first region in the cortex to receive this information is the primary auditory cortex (known as A1), which appears to be laid out in the same way as the cochlea itself (high frequencies at one end, low frequencies at the other). This is a pattern we see again and again in the brain – the primary sensory regions have a layout (or topology) which matches that of the sense receptors themselves.

The primary sensory regions, such as A1 (and its visual cousin, V1) are there to find what’s known as primitive spatial features in the sensory information. Each column in the region is connected to its own subset of the inputs, and it will fire or not depending on the number of its inputs which are active (or on). Thus, certain combinations of inputs will cause particular combinations of columns to become active. If you were able to watch the region’s columns from above in real time, you would see a fast-changing sequence of patterns, each of which has only a small percentage of columns active at any one time.

The match-up between the patterns of sound information entering A1 from the cochlea, and the resulting patterns of active columns, is of great interest. This match-up (or mapping) of inputs to activation is something which evolves over your entire life. What each column is doing is actually learning to respond in its own way to the set of inputs it receives at each moment. It is learning to recognise some patterns which arise again and again in the sounds you hear. For each pattern, only some of the inputs will be active, and the column learns by improving its connections to those inputs (on its proximal, or feedforward, dendrites) each time the pattern causes a successful firing. Inputs which are seldom active will be disimproved or neglected.

You can now see that the region has many columns, each of which is being fed a slightly different set of inputs, and each of which is tuning its connections to favour the patterns it most regularly receives (hears, in this case). At any given moment, then, the sound information entering the region causes a varying level of potential activation across the region, with each column displaying a certain amount of response to its input patterns. The columns with the most active inputs will fire first, and when they do this, they send out inhibitory signals to their near neighbours in the region. This acts to lower the neighbouring columns’ activity level even more, causing the best column to stand out in the activation pattern across the region.

The result is what is known in the theory as a *Sparse Distributed Representation* or SDR, which is a crucial idea for understanding how all this works. Each SDR is itself a spatial pattern, but it represents the features of the raw sound information which caused it. The SDR contains a cleaned-up version of the raw data, because only well-learned features will make it through the analysis. In addition, because of the sparseness caused by inhibition, only the most important feature patterns will survive to be seen in the SDR.

At this stage, the analysed information produced by A1 is sent up the hierarchy to a number of secondary regions. Some of these regions will start identifying features in the sounds which are speech-like in character, while others will be concerned with finding the location of the sound source, its music-like features, and so on. As the information is passed up the hierarchy, it is successively analysed for higher-level feature patterns.

Each region, in addition to learning to recognise and respond to spatial patterns, is also learning to recognise sequences of these patterns. This is another crucial idea in the HTM theory, and we’ll explain it in detail later on. We’ll continue now with an outline in the context of the hearing example.

Let’s move up the hierarchy now to a region which identifies speech-like sounds, or phonemes. This region will be mainly receiving SDR-encoded information, perhaps via a few intermediate regions, originating from the primary A1 region. In addition, it’ll have some connection with the part of the brain which creates and controls all the muscles used in speaking. The reason for this is that we learn to speak by listening to what happens when we make sounds using these muscles, and we keep adjusting the muscles until we learn the right patterns to get the sounds we expect to hear. So the “phoneme” region will itself be tuned to speech-like sounds based on both what it hears and what sounds we’ve learned to make in our own speech. This explains why, for example, native Japanese speakers have trouble distinguishing “l” and “r” sounds, or why English speakers have difficulty with tonal languages like Mandarin.

Each phoneme is actually caused by an intricate sequence of choreographed muscle movements, giving rise to a sequence of particular sound patterns. There are only a few dozen to a few hundred phonemes in each language, and so the sound sequences are easy to learn, and easy to distinguish against background noise. Note that there might be a large number of allowed sound patterns, but it’s the sequences which are key to identifying the phonemes.

So, we see that sequences of sound patterns can be combined to identify phonemes. In the same way (in a higher region), sequences of phonemes can be identified as particular words, sequences of words as phrases, and sequences of phrases as sentences. In this way, the hierarchy learns to hear entire sentences, as sequences of sequences of sequences, even though the sound data coming in was composed of thousands of patterns across 30,000 receptors.

And we can do the opposite. We can construct sentences from phrases we’ve learned, and send them down the hierarchy to generate the words, the phonemes, and the muscle movement sequences need to turn those phonemes into speech. This is the key to understanding how the Hierarchical Temporal Memory works.

**The Six Principles of the Neocortex**

Here are the six principles which Jeff Hawkins proposes are key to how the neocortex works.

1. On-line learning from streaming data

Everything we know comes ultimately from the constant stream of information coming in through our senses. At every moment, our brain is taking in this data, and adding to the learning done since before you were born. We do not store a copy of all this information – it’s coming in in huge quantities, we’ve limited capacity, and anyway lots of the individual data is not that important. So, we must do our best to turn it into something useful in real time.

1. Hierarchy of Memory Regions

The world is complex and ever changing, but it has structure. That structure is hierarchical in nature, and so your brain has a hierarchy which reflects that. The hierarchy serves to compress and abstract both in space and time, and allows us to throw around huge amounts of underlying information about the world very efficiently. It also allows us to find and exploit the connections between different, but related concepts and objects.

1. Sequence Memory

This is one of the key discoveries of the theory. Everything we experience arrives as a sequence of patterns over time. This is obvious in the case of speech, but it’s true of all sensory input.

Even our vision is not static, as we might suppose. We are constantly moving our eyes in tiny jumps (called saccades) and gathering little updates from around our field of view. What seems like a static image of the world in front of you is actually made from a memory of many, many sequences of brief glimpses, combined with some older memories from some time ago.

If I lay an object on the palm of my hand, I have only the barest idea what it is. I know how big it is by noticing the places where I can feel it on my hand, and I can tell its approximate shape and weight by feeling how hard it is at various points. Only when I move the object in my hand do I start to sense the details of its shape, texture, and features. It’s the sequence that contains the real information, since a certain object will create a specific sequence of perceptions when manipulated.

1. Sparse Distributed Representations

This is the second key discovery of the theory, and in recent years is becoming a widely accepted fact in mainstream neuroscience. SDR’s, to remind you, are representations in which we have only a few percent active columns or cells, all the rest being quiet. SDR’s, when they’re produced as they are in the brain, have a number of extremely powerful and useful properties, which are so important that we’ll devote a whole section to them below.

1. All Regions are Sensory and Motor

Again, this has only been discovered quite recently. We used to think that the sensory and motor parts of the brain were separate, but it turns out that every region has some kind of motor-related output. We don’t really understand how this works in detail, but it’s important to understand that the brain is learning sensorimotor models of the world – the movement is as important a part of the experiences as the new perception caused by that movement.

This reflects the fact that all sensory nerves are also “motor” nerves as well – they branch as they enter the brain and go to the classic sensory regions in the neocortex, in addition going to motor-related regions in the lower parts of the brain. The signals coming in convey one piece of information, but its meaning can be interpreted as both an item of sensory data (my leg is moving this way) and as an instruction for a corresponding movement to make in the near-future.

1. Attention

We have the ability to focus our awareness on certain bits of the hierarchy, ignoring much of what is happening elsewhere. If you look at the word “attention” above, you will notice that the rest of the page seems to fade out as you dwell on the word. Now, look at the letter “i” in that word, and you see that you are “zooming in” on that letter. Finally, you can just concentrate on the dot above the i, and you see that your attention can become almost microscopic. If you come across the word attention in a sentence, however, it’ll remain in focus for only a fraction of a second.

**Realising the Theory: The Cortical Learning Algorithm**

These six principles are, Hawkins claims, both necessary and sufficient for intelligence. In order to establish this, however, we’ll need to go beyond declaring the principles and create a detailed theoretical model of how these principles actually work, and then we can build computer systems based on these details.

In 2009, Hawkins and his team at Numenta made some significant breakthroughs in their work on HTM, which led to a new “sub-theory” called the Cortical Learning Algorithm (CLA). The CLA is a realisation in detail of these three central principles of HTM:

* On-line learning from streaming data
* Sequence Memory
* Sparse Distributed Representations

We’ll go into some detail on the CLA in the next Chapter.

In 2013 and early 2014, Hawkins has expanded the CLA to model two more of the six Principles:

* Hierarchy of Memory Regions
* All Regions are Sensory and Motor

This new theory, which I call Sensorimotor CLA, is a huge expansion and is still the subject of animated discussion on the community’s theory mailing list, so please treat my descriptions as preliminary and subject to revision.

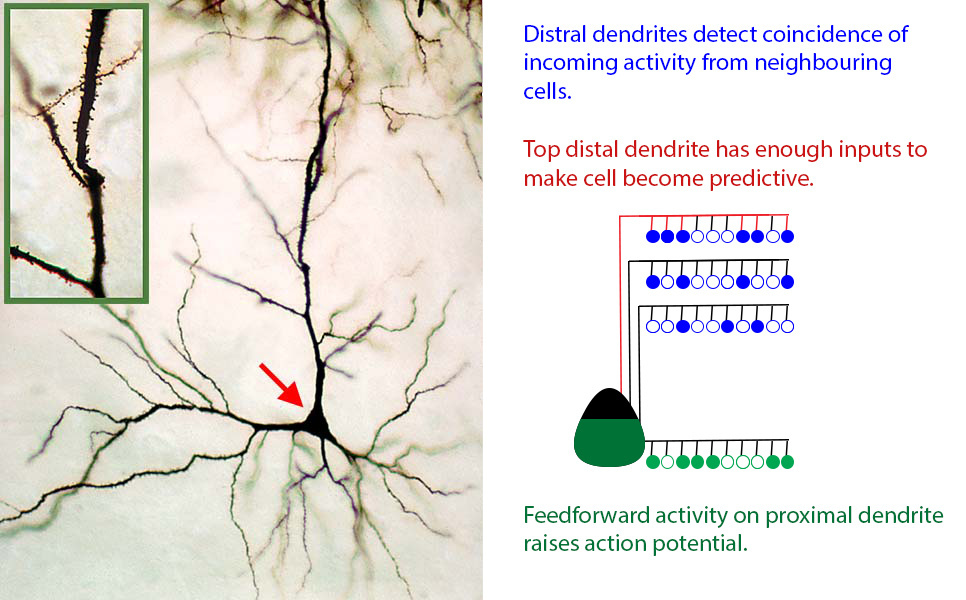
**Three: The Cortical Learning Algorithm**

The Cortical Learning Algorithm is a detailed mechanism for explaining the operation of a single-layer, single region of neocortex. This region is capable of learning sequences of spatial-temporal data, making predictions, and detecting anomalies in the data.

NuPIC and its commercial version, Grok, are realisations in software of the CLA, allowing us to observe how the model performs and also put it to work solving real-world problems.

**The CLA Model Neuron**

As described above, we know a lot about how individual neurons work in the neocortex. While each of the tens of billions of neurons in each of our brains is unique and very complex, for our purposes they can each be replaced with a standard neuron which looks like this:



Real (left) and CLA model (right) neurons

This model neuron is a significant simplification, but it captures the most important characteristics of real neurons. The key features are:

1. The neuron receives *feedforward inputs* at the proximal dendrites (bottom, green), each active input raising the action potential of the cell (green filling of the cell body).
2. If enough active signals come in at the proximal dendrites, the cell may become *active* and begin firing off its own signal.
3. Signals from other neurons (mostly nearby, mostly in the same region) are connected at a particular piece, or *segment* of the distal dendrite arbor (top, blue).
4. If enough signals appear on a segment, the segment (red, top segment) will cause the cell to become *predictive* of its own activity (red line).

This model neuron, while dramatically less complex and variable than a real one, is nevertheless a lot more complicated than the neurons found in other neural network systems.

When it comes to modelling the signals, we also use a simplified model of the real thing. The incoming signal is modelled by a “bit” - active or inactive, 1 or 0 - instead of the more realistic scalar value which would represent the signalling neuron’s firing rate or voltage.

The synapse (the junction between a signalling neuron and the receiving dendrite) is simplified too. It has a *permanence* which models the growth (or shrinkage) of the dendritic spine reaching out to the incoming neuron’s axon - this varies from 0.0 to 1.0. Secondly, the synapse is either “connected” or not - again a 1 or 0, based on whether the permanence is above or below a *threshold*. We will later see that this permanence is the key to learning - raising the permanence over time allows the CLA to “remember” a useful connection, while lowering it (due to disuse) allows it to “forget” that connection.

**The CLA Model of a Neocortical Region**

As we noted earlier, the neocortex is divided up into patches or regions of various sizes. A region is identified by the source of its feedforward inputs; for example, the primary auditory cortex A1 is identified as the receiver of the sound information coming in from the cochlea.

Each region in the real neocortex has five or six layers of neurons connected together in a very complex manner. The layered structure allows the region to perform some very complex processing, which combines information coming in from lower regions (or the senses), higher regions (feedback, “instructions” and attention control), and the information stored in the region itself (spatial-temporal memory).

At this stage in the development of the theory, we create a very simplified region which could be seen as a model of a single layer in the real neocortex. This consists of a two-dimensional grid of columns, each of which is a stack of our model neurons.

Before proceeding any further, we should take a look at a really important concept in HTM: Sparse Distributed Representations.

**Sparse Distributed Representations**

SDR’s are the currency of intelligence, both in the brain and in machine intelligence. Only SDR’s have the unique characteristics which allow us to learn, build and use robust models of the world based on experience and thought. This section explains why this way of representing a perception or an idea is so important.

SDR’s are *sparse*: in each example, there are many possible neurons, columns, or bits which could be on; however, most of them are inactive, quiet, or off. In NuPIC, for example, 2% of columns are used (typically, 40 out of 2,000 columns are active at any time). This may seem like a waste of neurons, but as we’ll see, there are some serious benefits to this.

SDR’s are *distributed*: the active cells are spread out across the region in an apparently random pattern. Over time, the patterns will visit all parts of the region and all cells will be involved in representing the activity of the region (if only for a small proportion of the time).

In addition, and very importantly, each active cell in an SDR has semantic meaning of some sort which comes from the structure in the inputs which caused it to become active. Each cell may be thought of as capturing some “feature” in the inputs, an example from vision being a vertical line in a certain place. But because the SDR is distributed, it’s important to note that no single cell is solely responsible for representing each feature - that responsibility is shared among a number of other active cells in each SDR.

**Important Properties of SDRs**

SDRs have some very important properties which, when combined, may prove crucial to their use both in the brain and in machine intelligence.

**Efficiency by Indexing**

In software, we can clearly store large SDRs very efficiently by just storing the locations of the active, or 1-bits, instead of storing all the zeros as well. In addition, we can compare two SDRs very quickly simply by comparing their active bits.

In the brain, the sparsity of neuronal activity allows for efficient use of energy and resources, by ensuring that only the strongest combinations of meaningful signals lead to activation, and that memory is used to store information of value.

**Subsampling and Semantic Overlap**

Because every active bit or neuron signifies some significant semantic information in the input, the mathematics of SDRs means that a subsampled SDR is almost as good as the whole representation. This means that any higher-level system which is “reading” the SDR needs only a subset of bits to convey the representation. In turn, this feature leads to greater fault-tolerance and robustness in the overall system.

The math also indicates that when two SDRs differ by a few bits, we can be highly confident that they are representing the same underlying thing, and even if this is not the case, the error is between two very semantically similar inputs.

**Unions of SDRs**

SDRs, like any bitmaps, can be combined in many ways. The most important combination in HTM is the (inclusive) OR operation, which turns on a bit if either or both inputs have that bit on. When the bitmaps are sparse, the OR (or union) will mostly contain bits from one or other of the inputs, with just a few bits shared commonly between them.

Given a set of SDRs (each representing one sensory experience, for example), the union will in a way represent the possibility of all members of the set. The union process is irreversible: all the bits are mixed together, so we cannot recover individual members. However, we can say with high confidence that a newly presented SDR is a member of the union.

This is very important for HTM, because it underlies the prediction process. When an HTM layer is making a prediction, it does so by combining all its predictions in a union of SDRs, and we can check if something is predicted by asking if its SDR is contained in this union.

**Creating a Sparse Distributed Representation: Pattern Memory (Spatial Pooling)**

Now that we have a basic understanding of SDRs, let’s look at how the CLA uses the structure of a layer to represent a sensory input as an SDR. This process has been called Spatial Pooling, because the layer is provided with a spatial pattern of active bits, and it is pooling the information from all these bits and forming a single SDR which (it is hoped) signifies the structure in the sensory data.

Later, we will see that this process is used for more complex things than spatial sensory patterns, so we’ll drop “Spatial Pooling” in favour of “Pattern Memory” in later discussions. For now, these are equivalent.

To make it simple for now, imagine that the neurons in each column are so similar that we can pretend each column contains only a single cell. We will add back the complexity later when we introduce sequence memory.

The layer now looks like a 2-dimensional grid (i.e., array) of single cells. Each cell has a proximal, or feedforward dendrite, which receives some input from the senses or a lower region in the hierarchy. Not every input bit is connected to every cell, so each cell will receive a subset of the input signal (we say it *subsamples* the input). We’ll assume that each cell receives a unique selection of subsampled bits. In addition, at any given moment, the synapses on a cell’s dendrites will each have some permanence, and this will further control how input signals are transmitted into the cell.

When an input appears, each active bit in the input signal will be transmitted only to a subset of cell dendrites in the layer (those which have that bit in their subsample or *potential pool*), and these signals will then continue only where the synapse permanence is sufficient to allow them through. Each cell thus acts as a kind of *filter*, receiving inputs from quite a small subsample of the incoming signal. If many bits from this subsample happen to be active, the cell will “charge up” with a high activation potential; on the other hand, if a cell has mostly off-bits in its receptive field, or if many of the synapses have low permanence, the total effect on the cell will be minimal, and its activation potential will be small or zero.

At this stage, we can visualise the layer as forming a *field* of activation potentials, with each cell in the grid possessing a level of activation potential (in the real brain, the cell membranes each have a real voltage, or potential). We now need to turn this into a binary SDR, and we do this using a strategy called “winner-takes-all” in machine learning, or “inhibition” in neuroscience and HTM.

The essential idea is to choose the cells with the highest potential to make active, and allow the lower potential cells be inhibited or suppressed by the winners. In HTM, we can do this either “globally”, by picking the highest n% potentials in a whole layer, or “locally”, by splitting the layer into neighbourhoods and choosing one or a few cells from each. The best choice depends on whether we wish to model the spatial structure of SDRs directly in the layout of the active cells - as happens in primary visual cortex V1, for example.

In the neocortex, each column of cells has a physical “sheath” of inhibitory cells, which are triggered if the column becomes active. These cells are very fast-acting, so they will almost instantly suppress the adjacent columns’ activity once triggered by a successful firing. In addition, these inhibitory cells are connected to each other, so a wave of inhibition will quickly spread out from an active column, leaving it isolated in an area of inactivity. This simple mechanism is sufficient to enforce almost all of the sparseness we see in the neocortex (there are many other mechanisms in the real cortex, but this is good enough for now).

The result is an SDR of active columns, which is the result of Spatial Pooling.

If this looks just like pattern recognition to you, you’d be correct. And you’d also be correct if you guessed how this process involves reinforcement learning, because this idea is common across many types of artificial neural networks. In most of these, there are numeric *weights* instead of our synapses, and these weights are adjusted so as to strengthen the best-matching connections between inputs and neurons. In the CLA, we instead adjust the *permanence* of the synapses. If the cell becomes active, we increase the permanence of any synapses which are attached to on-bits in the input, and we decrease slightly the permanence of synapses connected to the off-bits.

In this way (the mathematical properties of this kind of learning are discussed in detail elsewhere), each cell will gradually become better and better at recognising the subset of the inputs it sees most commonly over time, and so it’ll become more likely to become active when these, or similar, inputs appear again.

**Prediction Part I - Transition Memory**

The key to the power of CLA is in the combination of Pattern Memory with the ability to learn, “understand,” and predict how the patterns in the world change over time, and how these changes have a sequence-like structure which reflects structure in the real world. We now believe that all our models of reality (including our own internal world) are formed from memories of this nature.

We’ll now extend the theory a little to show how the CLA learns about individual steps in the changing sensory stream it experiences, and how these changes are chained together to form sequence memories. Again, we’ll need to postpone the full details in favour of a deep understanding of just this phase in the process.

**Active Columns to Active Cells**

Picking up the discussion from the last section on Pattern Memory (Spatial Pooling), remember that we pretended each column had only one cell when looking at the spatial patterns coming in from the senses. This was good enough to give us a way to recognise recurring structure in each pattern we saw, and form a “pooled” SDR on the layer which represented that pattern.

We now want to look at what happens when these patterns change (as all sensory input continually does), and how the CLA learns the structural information in these changes - we call them *transitions* - attempts to predict them and spot if something unexpected happens. We call this process **Transition Memory** (some people - including Jeff - call this *Temporal Memory*, but I like my term better!).

Pattern Memory in the last section used only the feedforward, or proximal, dendrites of each cell to look at the sensory inputs as they appeared. In learning transitions and making predictions, we use the remaining, distal dendrites to learn how one pattern leads onto the next. This is the key to the power of CLA in learning the spatiotemporal structure of the world.

In order to make this work, we must put back all the cells in each column, because it’s the way the cells behave which allows the transitions to be learned and predictions to be made. So, in each active column, we now have several dozen cells (in NuPIC we usually use 32) and we have to decide what is happening at this level of detail.

We’ll assume for now (and explain later) that in each active column, only one of the cells is active and the others are quiescent. Each active cell is sending out a signal on its axon (this is the definition of being active), and this axon has a branching structure a bit like the structure of the dendrites. Many axonal branches spread out horizontally in the layer and connect up with the dendrites of cells in neighbouring columns (perhaps spreading across distances many columns wide).

We’ll thus see (and can record) a spreading signal which connects currently active cells in some columns to many thousands of cells in many other columns. Some of the cells receiving these signals will happen to be connected to significant numbers of active cells, and some of these will happen to receive several signals close together on the same segment of a dendrite. If this happens, the segment will generate a *dendritic spike* (which is quite similar to the spike generated by an axon when it fires), and this spike will travel down the dendrite until it reaches the cell body. Each dendritic spike injects some small amount of electrical current into the cell body, and this raises the voltage across its membrane - a process called **depolarisation**.

Of course, many cells in the layer will receive little or no signals from the small number of active cells, and even if they do, the signals might appear on different branches and at different times. These signals will not appear close together enough to exceed a dendrite segment’s spiking threshold, and they will effectively be ignored. This means that, over the layer, we would be able to observe a second sparse pattern (or field) - this time of depolarisation caused by the signals between cells in the layer. Most cells will have zero depolarisation (they’ll be at a “resting potential”), but some cells will have a little and some will be highly depolarised.

This new pattern of depolarisation in the layer is a result of cells combining signals from the recently active pattern of cells we saw generated in the Spatial Pooling phase. The key to the whole Transition Memory process is that this second pattern is used to combine this signal from the recent past with the next sensory pattern which appears a short time later. Cells which are depolarised strongly by both the distal dendritic spikes (from cells in the recent past), and by the incoming, new sensory signal will have a head-start in reaching their firing threshold and will be the best bet to be among the top few percent picked to be in the next spatial pooling SDR.

We call cells which possess a high level of depolarisation due to horizontal signals **predictive**, since they will use this depolarisation to predict their own activity - if and when their expected sensory pattern arrives. If the prediction is correct, these cells will indeed become active, and they’ll do the same kind of reinforcement learning we saw in the previous section. Thus the layer learns how to associate the previous pattern with likely patterns coming next. This is the kernel of **Transition Memory**.

**Four: Numenta Platform for Intelligent Computing (NuPIC)**

**Turning Theory into Practice**

NuPIC is Numenta’s reference implementation of Jeff Hawkins’ HTM and CLA theories. It has evolved along with the theory since Jeff and Dileep George founded Numenta in 2005. NuPIC forms the core of Numenta’s new product, Grok, which provides a monitoring and alerting system for operators of Amazon Web Services instances. It’s also the centrepiece of a vibrant Open Source Project.

**How NuPIC works**

**Getting Data Into NuPIC - Encoders**

**Recognising an Input - the Spatial Pooler**

**Stringing Things Together - Temporal Prediction**

**Making Sense of the World - Classifier**

**Anomaly Detection and Probability**

**The NuPIC Community**

**NuPIC Architecture**

**nupic.core**

**Some Applications**

**Geospatial Encoder**

In July 2014, Numenta’s Chetan Surpur demoed and explained the details of a new encoder for NuPIC which creates Sparse Distributed Representations (SDRs) from GPS data. Apart altogether from the direct applications which this development immediately suggests, I believe that Chetan’s invention has a number of much more profound implications for NuPIC and even HTM in general. This section will explore a few of the most important of these. [Chetans’ demo](https://www.youtube.com/watch?v=KxxHo-FtKRo) and [a tutorial by Matt Taylor](https://www.youtube.com/watch?v=M4dD9wCQLkA) are available on Youtube.

**Mechanism**

The Geospatial Encoder takes as input a triple [Lat, Long, Speed] and returns a Sparse Distributed Representation (SDR) which uniquely identifies that position for the given speed. The speed is important because we want the “resolution” of the encoding to vary depending on how quickly the position is changing, and Chetan’s method does this very elegantly. The algorithm is quite simple. First, a 2D space (Lat, Long) is divided up (virtually) into squares of a given scale (a parameter provided for each encoder), so each square has an x and y integer co-ordinate (the Lat-Long pair is projected using a given projection scheme for convenient display on mapping software). This co-ordinate pair can then be used as a seed for a pseudorandom number generator (Python and numpy use the cross-platform Mersenne Twister MT19937), which is used to produce a real-valued order between 0 and 1, and a bit position chosen from the n bits in the encoding. These can be generated on demand for each square in the grid, always yielding the same results. To create the SDR for a given position and speed, the algorithm first converts the speed to a radius and forms a box of squares surrounding the position and calculates the pair [order, bit] for each square in the box. The top w squares (with the highest order) are chosen, and their bit values are used to choose the w active bits in the SDR.

**Initial Interpretation**

The first thing to say is that this encoder is an exemplar of transforming real-world data (location in the context of movement) into a very “SDR-like” SDR. It has the key properties we seek in an SDR encoder, in that semantically similar inputs will yield highly overlapping representations. It is robust to noise and measurement error in both space and time, and the representation is both unique (given a set scale parameter) and reproducible (by means of a cross-platform random number generator), independently of the order of data presentation. The reason for this “SDR-style” character is that the entire space of squares forms an infinite field of “virtual neurons”, each of which has some activation value (its order) and position in the input bit vector (its bit). The algorithm first sparsifies this representation by restricting its sampling subspace to a box of squares around the position, and then enforces the exact sparseness by picking the w squares using a competitive analogue of local inhibition.

**Random Spatial Neuron Field (Spatial Retina)**

This idea can be generalised to produce a “spatial retina” in n-dimensional space which provides a (statistically) unique SDR fingerprint for every point in the space. The SDRs specialise (or zoom in) when you reduce the radius factor, and generalise (or zoom out) when radius is increased. This provides a distance metric between two points which involves the interplay of spatial zoom and the fuzziness of overlap. Any two points will have identical SDRs (w bits of overlap) if you increase the radius sufficiently, and entirely disparate SDRs (0 bits overlap) if you zoom in sufficiently (down to the order of w\*scale). Since the Coordinate Encoder operates in a world of integer-indexed squares, we first need to transform each dimension using its own scale parameter (the Geospatial Encoder uses the same scale for each direction, but this is not necessary). We thus have a single, efficient, simple mechanism which allows HTM to navigate in any kind of spatial environment. This is, I believe, a really significant invention which has implications well beyond HTM and NuPIC. As Jeff and others mentioned during Chetan’s talk, this may be the mechanism underlying some animals’ ability to navigate using the Earth’s magnetic field. It is possible to envisage a (finite, obviously) field of real neurons which each have a unique response to position in the magnetic field. Humans have a similar ability to navigate, using sensory input to provide an activation pattern which varies over space and identifies locations. We combine whichever modalities work best (blind people use sound and memories of movement to compensate for impaired vision), and as long as the pipeline produces SDRs of an appropriate character, we can now see how this “just works”.

**Comparison with Random Distributed Scalar Encoder (RDSE)**

The Geospatial Encoder uses the more general Coordinate Encoder, which takes an n-dimensional integer vector and a radius and produces the corresponding SDR. It is easy to see how a 1D spatial encoder with a fixed speed would produce an SDR for arbitrary scalars, given an initial scale which would decide the maximum resolution of the encoder. This encoder would be an improved replacement for the RDSE, with the following advantages:

When encoding a value, the RDSE needs to encode all the values between existing encodings and the new value (so that the overlap guarantees are honoured). A 1D-Geo encoder can compute each value independently, saving significantly in time and memory footprint. In order to produce identical values for all inputs regardless of the order of presentation, the RDSE needs to “precompute” even more values in batches around a fixed “centre” (eg to compute f(23) starting at 0, we might have to compute [f(-30),…,f(30)]). Again, 1D-Geo scalar encoding computes each value uniquely and independently. Assuming scale (which decides the max resolution) is fixed, the 1D-Geo scalar encoding can compute encodings of variable resolution with semantic degradation by varying speed. The SDR for a value is exactly unique for the same speed, but changes gradually as speed is increased or decreased. The RDSE has no such property. This would strongly suggest that we can replace the RDSE with a 1D coordinate spatial encoder in NuPIC, and get all the above benefits without any compromise.

**Combination with Spatially-varying Data**

It is clear how you could combine this encoding scheme with data which varies by location, to create a richer idea of “order” in feeding the SDR generation algorithm. For example, you could combine random “order” with altitude or temperature data to choose the top w squares. Alternatively, the pure spatial bit signature of a location may be combined in parallel with the encoded values of scalar quantities found at the current location, so that an HTM system associatively learns the spatial structure of the given scalar field.

**Spatially Addressed Memory**

The Geospatial Encoder computes a symbolic SDR address for a spatial location, effectively a “name” or “word” for each place. The elements or alphabet of this encoding are simply random order activation values of nearby squares, so any more “real” semantic SDR-like activation pattern will do an even better job in computing spatial addresses. We use memories of spatial cues (literally, landmarks), emotional memories, maps, memories of moving within the space, textual directions, and so on to encode and reinforce these representations. This model explains why memory experts often use Memory Palaces (aka the Method of Loci) to remember long sequences of data items. They associate each item (or an imagined, memorable visual proxy) occupying a location in a very familiar spatial environment. It also explains the existence of “place neurons” in rodent hippocampi – these neurons are each participating in generating a spatial encoding similar in character to the Geospatial Encoder.

**Zooming, Panning, and Attention**

This is a wonderful model for how we “zoom in” or “zoom out” and perceive a continuously but smoothly varying model of the world. It also models how we can perceive gracefully degrading levels of detail depending on how much time or attention we pay to a perception. In this case, the “encoder” detailed here would be a subcortical structure or a thalamus-gated (attention controlled) input or relay between regions.

If we could find a mechanism in the brain which controls the size and position of a “window” of signals (akin to our variable box of squares), we would have a candidate for our ability to use attention to control spatial resolution and centre of focus. Such a mechanism may automatically arise from preferentially gating neurons at the edges of a “patch”, by virtue of the inhibition mechanism’s ability to smoothly alter the representation as inputs are added or removed. This mechanism would also explain boundary extension error, in which we “fill out” areas surrounding the physical boundaries of objects and images.

As explained in detail in her talk at the Royal Institute, Eleanor Maguire believes that the hippocampus is crucial for both this phenomenon and our ability to navigate in real space. As one of the brain components at the “top” of the hierarchies, the hippocampus may be the place where we can perform the crucial “zooming and panning” operations and where we manipulate spatial SDRs as suggested by the current discovery.

**Implementation Details**

The coordinate encoder has a deterministic, O(1), order-independent algorithm for computing both “order” and bit choice. One important issue is that the pseudorandom number is Python-specific, and so a Java encoder (which uses a different pseudorandom number generator) will produce completely different answers. The solution is to use the Python (and numpy) RNG, which is the Mersenne Twister MT19937, also used by default in numerous other languages. I believe it would be worth exploring using Perlin noise to generate the order and bit choice values. This would give you a) identical encodings across platforms, b) pseudorandom, uncorrelated values when the noise samples are far enough apart (eg when the inputs are integers as in this case), and c) smoothly changing values if you use very small step sizes.

Just one point about changing radius and its effect on the encoding. I’m very confident that the SDR is very robust to changes in radius, due to the sparsity of the SDRs. In other words, the overlap in an SDR at radius r with that at radius r’ (at the same GPS position) will be high, because you are only adding or removing an annulus around the same position (this will be similar to adding or removing a strip of squares when a small position change occurs).

**Links to the Demo and Encoder Code**

* [Chetan’s demo code (which is really comprehensive) on Github](https://github.com/numenta/nupic.geospatial).
* The [Geospatial Encoder code](https://github.com/numenta/nupic/blob/master/nupic/encoders/geospatial_coordinate.py) and the [Coordinate Encoder](https://github.com/numenta/nupic/blob/master/nupic/encoders/coordinate.py) are now part of NuPIC’s codebase.

**CEPT - The Cortical Engine for Processing Text**

**Five: Clortex - A Complementary Project**

**Introduction**

Clortex is a new design for HTM, designed to provide a basis for communication of and research into the theory, as well as exploration and development of new applications.

**Design Goals**

**Directly Analogous to HTM/CLA Theory**

In order to be a platform for demonstration, exploration and experimentation of Jeff Hawkins’ theories, the system must at all levels of relevant detail match the theory directly (ie 1:1). Any optimisations introduced may only occur following an effectively mathematical proof that this correspondence is maintained under the change.

There are several benefits to this requirement. Firstly, during development, this requirement provides a rigid and testable constraint on the options for implementation. With a good model of the theory in mind, we may proceed with confidence to use transparently analogous data structures and algorithms, leaving the question of computational performance for a later day.

Secondly, this requirement will ensure that the system at its heart remains a working implementation of the theory as it develops. In addition, because of this property, it will be directly usable by Jeff and any co-workers (including us) in extending and experimenting with new ideas in the theoretical space. This will enhance support for the new project, and encourage the HTM community to consider the new project as a parallel or alternative way to realise their own goals.

Thirdly, the software will provide a runnable explanation of the theory, with real working code (see next requirement) replacing the pseudocode and providing live imagery instead of diagrams (see later requirement).

Lastly, we feel that the theory deserves software of similar quality, and that the lack of this has slowed the realisation of the goals of all concerned. The development of a true analogue in software will pave the way for a rapid expansion in interest in the entire project. In particular, this will benefit anyone seeking to exploit the commercial advantages which the CLA offers.

**Transparently Understandable Implementation in Source Code**

All source code must at all times be readable by a non-developer. This can only be achieved if a person familiar with the theory and the models (but not a trained programmer) can read any part of the source code and understand precisely what it is doing and how it is implementing the algorithms.

This requirement is again deliberately very stringent, and requires the utmost discipline on the part of the developers of the software. Again, there are several benefits to this requirement.

Firstly, the extreme constraint forces the programmer to work in the model of the domain rather than in the model of the software. This constraint, by being adhered to over the lifecycle of the project, will ensure that the only complexity introduced in the software comes solely from the domain. Any other complexity introduced by the design or programming is known as incidental complexity and is the cause of most problems in software.

Secondly, this constraint provides a mechanism for verifying the first requirement. Any expert in the theory must be able to inspect the code for an aspect of the system and verify that it is transparently analogous to the theory.

Thirdly, anyone wishing to extend or enhance the software will be presented with no introduced obstacles, leaving only their level of understanding of the workings of the theory.

Finally, any bugs in the software should be reduced to breaches of this requirement, or alternatively, bugs in the theory.

**Directly Observable Data**

All relevant data structures representing the computational model must be directly observable and measurable at all times. A user must be able to inspect all this data and if required, present it in visual form.

This requirement ensures that the user of the platform can see what they’re doing at all times. The software is essentially performing a simulation of a simplified version of the neocortex as specified in the CLA, and the user must be able to directly observe how this simulation is progressing and how her choices in configuring the system might affect the computation.

The benefits of this requirement should be reasonably obvious. Two in particular: first, during development, a direct visual confirmation of the results of changes is a powerful tool; and secondly, this answers much of the representation problem, as it allows an observer to directly see how the models in the theory work, rather than relying on analogy.

**Sufficiently Performant**

The system must have performance sufficient to provide for rapid development of configurations suitable to a user task. In addition, the performance on large or complex data sets must be sufficient to establish that the system is succeeding in its task in principle, and that simply by scaling or optimising it can perform at production levels.

What this says is that the system must be a working prototype for how a more finely tuned or higher-performance equivalent will perform. Compute power and memory are cheap, and software can always be made faster relatively easily. The question a user has when using the system is primarily whether or not (and how well) the system can solve her problem, not whether it takes a few seconds or a few hours.

This constraint requires that the software infrastructure be designed so as to allow for significant raw performance improvements, both by algorithm upgrades and also by using concurrency and distribution when the user has the resources to scale the system.

**Useful Metrics**

The system must include functionality which allows the user to assess the effectiveness of configuration choices on the system at all relevant levels.

At present, NuPIC has some metrics but they are either difficult to understand and interpret, inappropriate, or both. The above requirement must be answered using metrics which have yet to be devised, so we have no further detail at this stage.

**Appropriate Platform**

The development language(s) and runtime platform must ensure ease of deployment, robust execution, easy maintenance and operation, reliability, extensibility, use in new contexts, portability, interoperability, and scaleable performance.

Quite a list, but each failure in the list reduces the potential mindshare of the software and raises fears for new adopters. Success in every item, along with the other requirements, ensures maximal usefulness and easy uptake by the rising ramp of the adoption curve.

**Design Philosophy**

It’s odd, but Clortex’ journey began when I followed a link to a [talk Jeff gave](http://gotocon.com/video#5) at GOTO Aarhus 2013, and decided to watch one, then two, and finally all three talks given by Russ Miles at the same event. If you’re only able to watch one, the one to watch is [Architectural Simplicity through Events](http://gotocon.com/video#56). In that talk, Russ outlines his axioms for building adaptable software:

**1. Your Software’s First Role is to be Useful**

Clearly, NuPIC is already useful, but there is a huge opportunity for Clortex to be useful in several new ways:

a) **As a Teaching Tool to help understand the CLA and its power**. HTM and CLA are difficult to understand at a deep level, and they’re very different from traditional Neural Networks in every way. A new design is needed to transparently communicate an intuitive view of CLA to layman, machine learning expert, and neuroscientist alike. The resulting understanding should be as clear to an intelligent and interested viewer as it is to Jeff himself.

b) **As a Research and Development platform for Machine Intelligence**. Jeff has recently added – literally – a whole set of layers to his theory, involving a new kind of temporal pooling, sensorimotor modelling, multilayer regions, behaviour, subcortical connections, and hierarchy. This is all being done with thought experiments, whiteboards, pen and paper, and slides. We’ll see this in software sometime, no doubt, but that process has only begun. A new system which allows many of these ideas to be directly expressed in software and tested in real time will accelerate the development of the theory and allow many more people to work on it.

c) **As a Production Platform for new Use Cases**. NuPIC is somewhat optimised for a certain class of use cases – producing predictions and detecting anomalies in streaming machine-generated numerical data. It’s also been able to demonstrate capabilities in other areas, but there is a huge opportunity for a new design to allow entirely new types of information to be handled by HTM and CLA techniques. These include vision, natural language, robotics, and many other areas to which traditional AI and ML techniques have been applied with mixed results. A new design, which emphasises adaptability, flexibility, scaleability and composability, will allow CLA to be deployed at whatever scale (in terms of hierarchy, region size, input space etc as well as machine resources) is appropriate to the task.

**2. The best software is that which is not needed at all**

Well, we have our brains, and the whole point of this is to build software which uses the principles of the brain. On the other hand, we can minimise over-production by only building the components we need, once we understand how they work and how they contribute to the overall design. Clortex embraces this using a design centred around immutable data structures, surrounded by a growing set of transforming functions which work on that data.

**3. Human Comprehension is King**

This axiom is really important for every software project, but so much more so when the thing you’re modelling is so difficult to understand for many. The key to applying this axiom is to recognise that the machine is only the second most important audience for your code – the most important being other humans who will interact with your code as developers, researchers, and users. Clortex has as its #1 requirement the need to directly map the domain – Jeff’s theory of the neocortex – and to maintain that mapping at all costs. This alone would justify building Clortex for me.

**4. Machine Sympathy is Queen**

This would seem to contradict Axiom 3, but the use of the word “Queen” is key. Any usable system must also address the machine environment in which it must run, and machine sympathy is how you do that. Clortex’ design is all about turning constraints into synergies, using the expressive power and hygiene of Clojure and its immutable data structures, the unique characteristics of the Datomic database system, and the scaleability and portability characteristics of the Java Virtual Machine. Clortex will run on Raspberry Pi, a version of which will run in browsers and phones, yet it will scale layers and hierarchies across huge clusters to deliver real power and test the limits of HTM and CLA in production use.

**5. Software is a Process of R&D**

This is obviously the case when you’re building software based on an evolving theory of how the brain performs analogous functions. Russ’ key point here is that our work always involves unknowns, and our software and processes must be designed in such a way as not to slow us down in our R&D work. Clortex is designed as a set of loosely coupled, interchangeable components around a group of core data structures, and communicating using simple, immutable data.

**6. Software Development is an Extremely Challenging Intellectual Pursuit**

Again, this is so true in this case, but the huge payoff you can derive if you can come up with a design which matches the potential of the CLA is hard to beat. I hope that Clortex can meet this most extreme of challenges.

I met Russ in Spring 2014 when he visited Dublin to give a talk, and we had a great discussion (as ever) in the pub afterwards, about Simplicity, Antifragile Software and, of course, how HTM, and understanding the brain, is going to change the world. Russ is putting together his ideas and methods in his own Leanpub book, [Antifragile Software](https://leanpub.com/antifragilesoftware?a=3gYY0cd_58RxWCcx7Ukwgl), which I’d strongly encourage you to read.

**Platform - Clojure, Functional Programming on the Java Virtual Machine**

NuPIC is a hybrid, written in Python and partly re-implemented (nupic.core) in C++ for performance and cross-platform applicability. Python is a great language, but it’s at heart an Object-oriented (OO) language, and NuPIC has a complex OO-style structure as a result. As I was investigating possible platforms for a new HTM system, it became clearer and clearer to me that the world of software is undergoing a revolution, shedding decades of distraction and complexity in favour of a return to its roots: Functional Programming.

Clojure is a relatively young language (first released in 2008), but in fact it’s an update of the second-oldest popular language ever - LISP - which dates back to 1958.

It turns out that Rich Hickey, Clojure’s author, is also a neuroscience nut.

**Architecture - Russ Miles’ Life Preserver**

Russ’ idea for Antifragile Software is called the Life Preserver (aka the Lifebelt). The design was originally conceived to provide a simple answer to the question “where do I put this?” when architecting a software system. Russ’ answer is that each module, object, or subsystem in a design belongs either in the centre, or *core*, or else in the ring around the core, in an *integration* domain.

After many weeks of thought (simplicity is not easy!), the architecture of Clortex practically designed itself. The core of Clortex is simply a big data structure, which contains all the layers (called *patches* in Clortex) of neurons, each layer organised into columns as in the theory. Each neuron has a proximal dendrite and a number of distal dendrites, and each dendrite contains a set of simple synapses which connect with other neurons.

This data structure is stored in a new kind of database - Datomic. I’ll go into more detail about Datomic later, but for now here are some of the key attributes which Datomic brings to Clortex:

**What’s Next?**

**Jeff’s New Theory - Sensorimotor HTM**

In late 2013 and early 2014, Jeff has added a new dimension to the HTM theory by including behaviour in his picture of the function of the neocortex.

Jeff recently talked about a sensorimotor extension for his Cortical Learning Algorithm (CLA). This extension involves Layer 4 cells learning to predict near-future sensorimotor inputs based on the current sensory input and a copy of a related motor instruction. This section briefly describes an idea which can explain both the mechanism, and several useful properties, of this phenomenon. It is both a philosophical and a neuroscientific idea, which serves to explain our experience of cognition, and simultaneously explains an aspect of the functioning of the cortex.

In essence, Jeff’s new idea is based on the observation that Layer 4 cells in a region receive information about a part of the current sensory (afferent, feedforward) inputs to the region, along with a copy of related motor command activity. The idea is that Layer 4 combines these to form a prediction of the next set of sensory inputs, having previously learned the temporal coincidence of the sensory transition and the effect of executing the motor command.

One easily visualised example is that a face recognising region, currently perceiving a right eye, can learn to predict seeing a left eye when a saccade to the right is the motor command, and/or a nose when a saccade to the lower right is made, etc. Jeff proposes that this is used to form a stable representation of the face in Layer 3, which is receiving the output of these Layer 4 cells.

I believe that the “motor command” represents either a real motor command to be executed, which will cause the predicted change in sensory input, or else the analogous “change in the world” which would have the same transitional sensory effect. The latter would represent, in the above example, the person whose face is seen, moving her own head in the opposite direction, and presenting an eye or nose to the observer while the observer is passive.

In the case of speech recognition, the listener uses her memory of how to make the next sound to predict which sounds the speaker is likely to make next. At the same time, the speaker is using his memory of the sound he expects to make to perform fine control over his motor behaviour.

Another example is the experience of sitting on a stationary train when another train begins to move out of the station. The stationary observer often gets the feeling that she is in fact moving and that the other train is not (and a person in the other train may have the opposite perception – that he is stationary and the first person’s train is the one which is moving).

The colloquial term for this idea is the notion of a “mirror cell”. This article claims that so-called “mirror cells” are pervasive at all levels of the cortex and serve to explain exactly why every region of cortex produces “motor commands” in the processing of what is usually considered pure sensory information.

In this way, the cortex is creating a truly integrated sensorimotor model, which not only contains and explains the temporal structure of the world, but also stores and provides the “means of construction” of that temporal structure in terms of how it can be generated (either by the action of the observer interacting with the world, or by the passive observation of the external action of some cause in the world).

This idea also provides an explanation for the learning power of the cortex. In learning to perceive the world, we need to provide – literally – a “motivation” for every observed event in the world, as either the result of our action or by the occurrence of a precisely mirrored action caused externally. At a higher cognitive level, this explains why the best way to learn anything is to “do it yourself” – whether it’s learning a language or proving a theorem. Only when we have constructed both an active and a passive sensorimotor model of something do we possess true understanding of it.

Finally, this idea explains why some notions are hard to “get” at times – this model requires a listener or learner not just to imagine the sensory perception or cognitive “snapshot” of an idea, but the events or actions which are involved in its construction or establishment in the world.

**Third-party Projects Based on HTM**

**Appendix 1 - NuPIC Resources**

**Appendix 2: Clortex Resources**

**Appendix 3: Further Reading and Resources**

**HTM Projects and Information**

**Online Courses**

**Appendix 4: Mathematics of HTM**

This article describes some of the mathematics underlying the theory and implementations of Jeff Hawkins’ Hierarchical Temporal Memory (HTM), which seeks to explain how the neocortex processes information and forms models of the world.

**The HTM Model Neuron - Pattern Memory (aka Spatial Pooling)**

We’ll illustrate the mathematics of HTM by describing the simplest operation in HTM’s Cortical Learning Algorithm: Pattern Memory, also known as Spatial Pooling, forms a Sparse Distributed Representation from a binary input vector. We begin with a layer (a 1- or 2-dimensional array) of single neurons, which will form a pattern of activity aimed at efficiently representing the input vectors.

**Feedforward Processing on Proximal Dendrites**

The HTM model neuron has a single *proximal dendrite*, which is used to process and recognise feedforward or afferent inputs to the neuron. We model the entire feedforward input to a cortical layer as a bit vector {\mathbf x}_ {\textrm{ff}}\in\lbrace{0,1}\rbrace^ {n_ {\textrm{ff}}}, where n_ {\textrm{ff}} is the width of the input.

The dendrite is composed of n_ s*synapses* which each act as a binary gate for a single bit in the input vector. Each synapse has a *permanence* p_ i\in{[0,1]} which represents the size and efficiency of the dendritic spine and synaptic junction. The synapse will transmit a 1-bit (or on-bit) if the permanence exceeds a threshold \theta_ i (often a global constant \theta_ i = \theta = 0.2). When this is true, we say the synapse is *connected*.

Each neuron samples n_ s bits from the n_ {\textrm{ff}} feedforward inputs, and so there are {n_ {\textrm{ff}}}\choose{n_ {s}} possible choices of input for a single neuron. A single proximal dendrite represents a projection \pi_ j:\lbrace{0,1}\rbrace^ {n_ {\textrm{ff}}}\rightarrow\lbrace{0,1}\rbrace^ {n_ s}, so a population of neurons corresponds to a set of subspaces of the sensory space. Each dendrite has an input vector {\mathbf x}_ j=\pi_ j({\mathbf x}_ {\textrm{ff}}) which is the projection of the entire input into this neuron’s subspace.

A synapse is connected if its permanence p_ i exceeds its threshold \theta_ i. If we subtract {\mathbf p}-{\vec\theta}, take the elementwise sign of the result, and map to \lbrace{0,1}\rbrace, we derive the binary connection vector {\mathbf c}_ j for the dendrite. Thus:

c_ i=(1 + sgn(p_ i-\theta_ i))/2

The dot product o_ j({\mathbf x})={\mathbf c}_ j\cdot{\mathbf x}_ j now represents the *feedforward overlap* of the neuron with the input, ie the number of connected synapses which have an incoming activation potential. Later, we’ll see how this number is used in the neuron’s processing.

The elementwise product {\mathbf o}_ j={\mathbf c}_ j\odot{\mathbf x}_ j is the vector in the neuron’s subspace which represents the input vector {\mathbf x}_ {\textrm{ff}} as “seen” by this neuron. This is known as the *overlap vector*. The length o_ j = \lVert{\mathbf o}_ j\rVert_ {\ell_ 1} of this vector corresponds to the extent to which the neuron recognises the input, and the direction (in the neuron’s subspace) is that vector which has on-bits shared by both the connection vector and the input.

If we project this vector back into the input space, the result \mathbf{\hat{x}}_ j =\pi^ {-1}({\mathbf o}_ j) is this neuron’s approximation of the part of the input vector which this neuron matches. If we add a set of such vectors, we will form an increasingly close approximation to the original input vector as we choose more and more neurons to collectively represent it.

**Sparse Distributed Representations (SDRs)**

We now show how a layer of neurons transforms an input vector into a sparse representation. From the above description, every neuron is producing an estimate \mathbf{\hat{x}}_ j  of the input {\mathbf x}_ {\textrm{ff}}, with length o_ j\ll n_ {\textrm{ff}} reflecting how well the neuron represents or recognises the input. We form a sparse representation of the input by choosing a set Y_ {\textrm{SDR}} of the top n_ {\textrm{SDR}}=sN neurons, where N is the number of neurons in the layer, and s is the chosen sparsity we wish to impose (typically s=0.02=2\%).

The algorithm for choosing the top n_ {\textrm{SDR}} neurons may vary. In neocortex, this is achieved using a mechanism involving cascading inhibition: a cell firing quickly (because it depolarises quickly due to its input) activates nearby inhibitory cells, which shut down neighbouring excitatory cells, and also trigger further nearby inhibitory cells, spreading the inhibition outwards. This type of *local inhibition* can also be used in software simulations, but it is expensive and is only used where the design involves spatial topology (ie where the semantics of the data is to be reflected in the position of the neurons). A more efficient *global inhibition* algorithm - simply choosing the top n_ {\textrm{SDR}} neurons by their depolarisation values - is often used in practise.

If we form a bit vector {\mathbf y}_ {\textrm{SDR}}\in\lbrace{0,1}\rbrace^ N\textrm{ where } y_ j = 1 \Leftrightarrow j \in Y_ {\textrm{SDR}}, we have a function which maps an input {\mathbf x}_ {\textrm{ff}}\in\lbrace{0,1}\rbrace^ {n_ {\textrm{ff}}} to a sparse output {\mathbf y}_ {\textrm{SDR}}\in\lbrace{0,1}\rbrace^ N, where the length of each output vector is \lVert{\mathbf y}_ {\textrm{SDR}}\rVert_ {\ell_ 1}=sN \ll N.

The reverse mapping or estimate of the input vector by the set Y_ {\textrm{SDR}} of neurons in the SDR is given by the sum:

\mathbf{\hat{x}} = \sum\limits_ {j \in Y_ {\textrm{SDR}}}{{\mathbf{\hat{x}}}_ j} = \sum\limits_ {j \in Y_ {\textrm{SDR}}}{\pi_ j^ {-1}({\mathbf o}_ j)} = \sum\limits_ {j \in Y_ {\textrm{SDR}}}{\pi_ j^ {-1}({\mathbf c}_ j\odot{\mathbf x}_ j)}= \sum\limits_ {j \in Y_ {\textrm{SDR}}}{\pi_ j^ {-1}({\mathbf c}_ j \odot \pi_ j({\mathbf x}_ {\textrm{ff}}))}= \sum\limits_ {j \in Y_ {\textrm{SDR}}}{\pi_ j^ {-1}({\mathbf c}_ j) \odot {\mathbf x}_ {\textrm{ff}}}

**Matrix Form**

The above can be represented straightforwardly in matrix form. The projection \pi_ j:\lbrace{0,1}\rbrace^ {n_ {\textrm{ff}}} \rightarrow\lbrace{0,1}\rbrace^ {n_ s}  can be represented as a matrix \Pi_ j \in \lbrace{0,1}\rbrace^ {{n_ s} \times\ n_ {\textrm{ff}}} .

Alternatively, we can stay in the input space \mathbb{B}^ {n_ {\textrm{ff}}}, and model \pi_ j as a vector \vec\pi_ j =\pi_ j^ {-1}(\mathbf 1_ {n_ s}), ie where \pi_ {j,i} = 1 \Leftrightarrow (\pi_ j^ {-1}(\mathbf 1_ {n_ s}))_ i = 1.

The elementwise product \vec{x_ j} =\pi_ j^ {-1}(\mathbf x_ {j}) = \vec{\pi_ j} \odot {\mathbf x_ {\textrm{ff}}} represents the neuron’s view of the input vector x_ {\textrm{ff}}.

We can similarly project the connection vector for the dendrite by elementwise multiplication: \vec{c_ j} =\pi_ j^ {-1}(\mathbf c_ {j}) , and thus \vec{o_ j}(\mathbf x_ {\textrm{ff}}) = \vec{c_ j} \odot \mathbf{x}_ {\textrm{ff}} is the overlap vector projected back into \mathbb{B}^ {n_ {\textrm{ff}}}, and the dot product o_ j(\mathbf x_ {\textrm{ff}}) = \vec{c_ j} \cdot \mathbf{x}_ {\textrm{ff}} gives the same overlap score for the neuron given \mathbf x_ {\textrm{ff}} as input. Note that \vec{o_ j}(\mathbf x_ {\textrm{ff}}) =\mathbf{\hat{x}}_ j , the partial estimate of the input produced by neuron j.

We can reconstruct the estimate of the input by an SDR of neurons Y_ {\textrm{SDR}}:

\mathbf{\hat{x}}_ {\textrm{SDR}} = \sum\limits_ {j \in Y_ {\textrm{SDR}}}{{\mathbf{\hat{x}}}_ j} = \sum\limits_ {j \in Y_ {\textrm{SDR}}}{\vec o}_ j = \sum\limits_ {j \in Y_ {\textrm{SDR}}}{{\vec c}_ j\odot{\mathbf x_ {\textrm{ff}}}} = {\mathbf C}_ {\textrm{SDR}}{\mathbf x_ {\textrm{ff}}}

where {\mathbf C}_ {\textrm{SDR}} is a matrix formed from the {\vec c}_ j for j \in Y_ {\textrm{SDR}}.

**Optimisation Problem**

We can now measure the distance between the input vector \mathbf x_ {\textrm{ff}} and the reconstructed estimate \mathbf{\hat{x}}_ {\textrm{SDR}} by taking a norm of the difference. Using this, we can frame learning in HTM as an optimisation problem. We wish to minimise the estimation error over all inputs to the layer. Given a set of (usually random) projection vectors \vec\pi_ j for the N neurons, the parameters of the model are the permanence vectors \vec{p}_ j, which we adjust using a simple Hebbian update model.

The update model for the permanence of a synapse p_ i on neuron j is:

p_ i^ {(t+1)} =
\begin{cases}
(1+\delta_ {inc})p_ i^ {(t)} & \text{if } j \in Y_ {\textrm{SDR}}\text{, }(\mathbf x_ j)_ i=1\text{ and } p_ i^ {(t)} \ge \theta_ i \\
(1-\delta_ {dec})p_ i^ {(t)} & \text{if } j \in Y_ {\textrm{SDR}} \text{ and (}(\mathbf x_ j)_ i=0 \text{ or }p_ i^ {(t)} < \theta_ i \text{)} \\
p_ i^ {(t)} & \text{otherwise} \\
\end{cases}

This update rule increases the permanence of active synapses, those that were connected to an active input when the cell became active, and decreases those which were either disconnected or received a zero when the cell fired. In addition to this rule, an external process gently boosts synapses on cells which either have a lower than target rate of activation, or a lower than target average overlap score.

I do not yet have the proof that this optimisation problem converges, or whether it can be represented as a convex optimisation problem. I am confident such a proof can be easily found. Perhaps a kind reader who is more familiar with a problem framed like this would be able to confirm this. I’ll update this post with more functions from HTM in coming weeks.

**Transition Memory - Making Predictions**

In Part One, we saw how a layer of neurons learns to form a Sparse Distributed Representation (SDR) of an input pattern. In this section, we’ll describe the process of learning temporal sequences.

We showed in part one that the HTM model neuron learns to recognise subpatterns of feedforward input on its proximal dendrites. This is somewhat similar to the manner by which a Restricted Boltzmann Machine can learn to represent its input in an unsupervised learning process. One distinguishing feature of HTM is that the evolution of the world over time is a critical aspect of what, and how, the system learns. The premise for this is that objects and processes in the world persist over time, and may only display a portion of their structure at any given moment. By learning to model this evolving revelation of structure, the neocortex can more efficiently recognise and remember objects and concepts in the world.

**Distal Dendrites and Prediction**

In addition to its one proximal dendrite, a HTM model neuron has a collection of *distal* (far) dendrite segments (or simply *dendrites*), which gather information from sources other than the feedforward inputs to the layer. In some layers of neocortex, these dendrites combine signals from neurons in the same layer as well as from other layers in the same region, and even receive indirect inputs from neurons in higher regions of cortex. We will describe the structure and function of each of these.

The simplest case involves distal dendrites which gather signals from neurons within the same layer.

In Part One, we showed that a layer of N neurons converted an input vector \mathbf x \in \mathbb{B}^ {n_ {\textrm{ff}}} into a SDR \mathbf{y}_ {\textrm{SDR}} \in \mathbb{B}^ {N}, with length\lVert{\mathbf y}_ {\textrm{SDR}}\rVert_ {\ell_ 1}=sN \ll N, where the sparsity s is usually of the order of 2% (N is typically 2048, so the SDR \mathbf{y}_ {\textrm{SDR}} will have 40 active neurons).

The layer of HTM neurons can now be extended to treat its own activation pattern as a separate and complementary input for the next timestep. This is done using a collection of distal dendrite segments, which each receive as input the signals from other neurons in the layer itself. Unlike the proximal dendrite, which transmits signals directly to the neuron, each distal dendrite acts individually as an active coincidence detector, firing only when it receives enough signals to exceed its individual threshold.

We proceed with the analysis in a manner analogous to the earlier discussion. The input to the distal dendrite segment k at time t is a sample of the bit vector \mathbf{y}_ {\textrm{SDR}}^ {(t-1)}. We have n_ {ds} distal synapses per segment, a permanence vector \mathbf{p}_ k \in [0,1]^ {n_ {ds}} and a synapse threshold vector \vec{\theta}_ k \in [0,1]^ {n_ {ds}}, where typically \theta_ i = \theta = 0.2 for all synapses.

Following the process for proximal dendrites, we get the distal segment’s connection vector \mathbf{c}_ k:

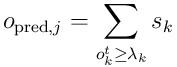
c_ {k,i}=(1 + sgn(p_ {k,i}-\theta_ {k,i}))/2

The input for segment k is the vector \mathbf{y}_ k^ {(t-1)} = \phi_ k(\mathbf{y}_ {\textrm{SDR}}^ {(t-1)}) formed by the projection \phi_ k:\lbrace{0,1}\rbrace^ {N-1}\rightarrow\lbrace{0,1}\rbrace^ {n_ {ds}} from the SDR to the subspace of the segment. There are {N-1}\choose{n_ {ds}} such projections (there are no connections from a neuron to itself, so there are N-1 to choose from).

The overlap of the segment for a given \mathbf{y}_ {\textrm{SDR}}^ {(t-1)} is the dot product o_ k^ t = \mathbf{c}_ k\cdot\mathbf{y}_ k^ {(t-1)}. If this overlap exceeds the threshold \lambda_ k of the segment, the segment is active and sends a dendritic spike of size s_ k to the neuron’s cell body.

This process takes place before the processing of the feedforward input, which allows the layer to combine contextual knowledge of recent activity with recognition of the incoming feedforward signals. In order to facilitate this, we will change the algorithm for Pattern Memory as follows.

Each neuron j begins a timestep t by performing the above processing on its {n_ {\textrm{dd}}} distal dendrites. This results in some number 0\ldots{n_ {\textrm{dd}}}of segments becoming active and sending spikes to the neuron. The total predictive activation potential is given by:



The predictive potential is combined with the overlap score from the feedforward overlap coming from the proximal dendrite to give the total activation potential:

a_ j^ t=\alpha_ j o_ {\textrm{ff},j} + \beta_ j o_ {\textrm{pred},j}

and these a_ j potentials are used to choose the top neurons, forming the SDR Y_ {\textrm{SDR}} at time t. The mixing factors \alpha_ k and \beta_ k are design parameters of the simulation.

**Learning Predictions**

We use a very similar learning rule for distal dendrite segments as we did for the feedforward inputs:

{p_ {i,j}}^ {(t+1)} =
\begin{cases}
(1+\sigma_ {inc})p_ i^ {(t)} & \text {if cell } j \text{ active, segment } k \text{ active, synapse } i \text{ active} \\
(1-\sigma_ {dec})p_ i^ {(t)} & \text {if cell } j \text { active, segment } k \text{ active, synapse } i \text{ not active} \\
p_ i^ {(t)} & \text{otherwise} \\
\end{cases}

Again, this reinforces synapses which contribute to activity of the cell, and decreases the contribution of synapses which don’t. A boosting rule, similar to that for proximal synapses, allows poorly performing distal connections to improve until they are good enough to use the main rule.

**Interpretation**

We can now view the layer of neurons as forming a number of representations at each timestep. The field of predictive potentials o_ {\textrm{pred},j}can be viewed as a map of the layer’s confidence in its prediction of the next input. The field of feedforward potentials o_ {\textrm{ff},j} can be viewed as a map of the layer’s recognition of current reality. Combined, these maps allow for prediction-assisted recognition, which, in the presence of temporal correlations between sensory inputs, will improve the recognition and representation significantly.

We can quantify the properties of the predictions formed by such a layer in terms of the mutual information between the SDRs at time tand t+1. I intend to provide this analysis as soon as possible, and I’d appreciate the kind reader’s assistance if she could point me to papers which might be of help.

A layer of neurons connected as described here is a Transition Memory, and is a kind of first-order memory of temporally correlated transitions between sensory patterns. This kind of memory may only learn one-step transitions, because the SDR is formed only by combining potentials one timestep in the past with current inputs.

Since the neocortex clearly learns to identify and model much longer sequences, we need to modify our layer significantly in order to construct a system which can learn high-order sequences. This is the subject of the next part of this Appendix.

Note: For brevity, I’ve omitted the matrix treatment of the above. See Part One for how this is done for Pattern Memory; the extension to Transition Memory is simple but somewhat arduous.

**Draft White Paper 2014**

**Chapter 1 - Hierarchical Temporal Memory (HTM) Overview**

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

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

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

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

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

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

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

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

**HTM principles**

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

**Hierarchy**

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

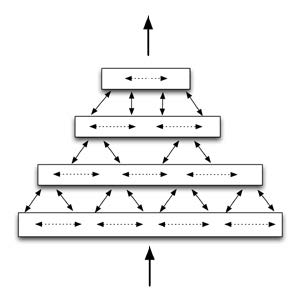


Figure 1.1: Simplified diagram of four HTM regions arranged in a four-level hierarchy, communicating information within levels, between levels, and to/from outside the hierarchy

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

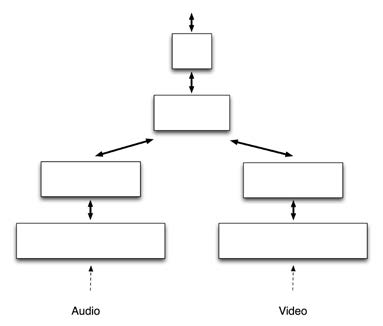


Figure 1.2: Converging networks from different sensors

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

As another example, consider that when you learn a new word, you don’t need to relearn letters, syllables, or phonemes. Sharing representations in a hierarchy also leads to generalization of expected behavior. When you see a new animal, if you see a mouth and teeth you will predict that the animal eats with his mouth and that it might bite you. The hierarchy enables a new object in the world to inherit the known properties of its sub-components.

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

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

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

**Regions**

The notion of regions wired in a hierarchy comes from biology. The neocortex is a large sheet of neural tissue about 2mm thick. Biologists divide the neocortex into different areas or “regions” primarily based on how the regions connect to each other. Some regions receive input directly from the senses and other regions receive input only after it has passed through several other regions. It is the region-to-region connectivity that defines the hierarchy.

All neocortical regions look similar in their details. They vary in size and where they are in the hierarchy, but otherwise they are similar. If you take a slice across the 2mm thickness of a neocortical region, you will see six layers, five layers of cells and one non-cellular layer (there are a few exceptions but this is the general rule). Each layer in a neocortical region has many interconnected cells arranged in columns. HTM regions also are comprised of a sheet of highly interconnected cells arranged in columns. “Layer 3” in neocortex is one of the primary feed-forward layers of neurons. The cells in an HTM region are roughly equivalent to the neurons in layer 3 in a region of the neocortex.

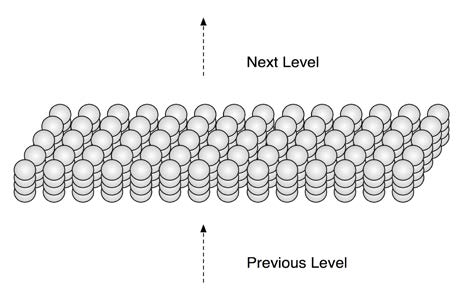


Figure 1.3: A section of an HTM region. HTM regions are comprised of many cells. The cells are organized in a two dimensional array of columns. This figure shows a small section of an HTM region with four cells per column. Each column connects to a subset of the input and each cell connects to other cells in the region (connections not shown). Note that this HTM region, including its columnar structure, is equivalent to one layer of neurons in a neocortical region.

Although an HTM region is equivalent to only a portion of a neocortical region, it can do inference and prediction on complex data streams and therefore can be useful in many problems.

**Sparse Distributed Representations**

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

HTM regions also use sparse distributed representations. In fact, the memory mechanisms within an HTM region are dependent on using sparse distributed representations, and wouldn’t work otherwise. The input to an HTM region is always a distributed representation, but it may not be sparse, so the first thing an HTM region does is to convert its input into a sparse distributed representation.

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

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

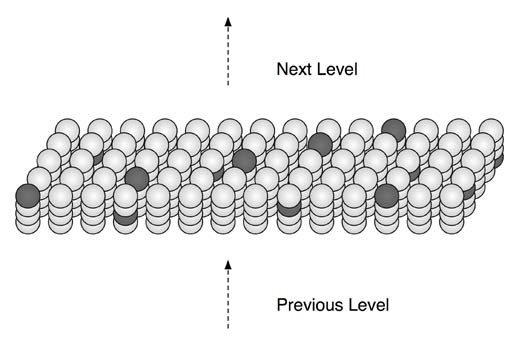


Figure 1.4: An HTM region showing sparse distributed cell activation

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

**The role of time**

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

Let’s start with inference. Without using time, we can infer almost nothing from our tactile and auditory senses. For example if you are blindfolded and someone places an apple in your hand, you can identify what it is after manipulating it for just a second or so. As you move your fingers over the apple, although the tactile information is constantly changing, the object itself – the apple, as well as your high-level percept for “apple” – stays constant. However, if an apple was placed on your outstretched palm, and you weren’t allowed to move your hand or fingers, you would have great difficulty identifying it as an apple rather than a lemon.

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

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

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

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

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

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

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

**Learning**

An HTM region learns about its world by finding patterns and then sequences of patterns in sensory data. The region does not “know” what its inputs represent; it works in a purely statistical realm. It looks for combinations of input bits that occur together often, which we call spatial patterns. It then looks for how these spatial patterns appear in sequence over time, which we call temporal patterns or sequences. If the input to the region represents environmental sensors on a building, the region might discover that certain combinations of temperature and humidity on the north side of the building occur often and that different combinations occur on the south side of the building. Then it might learn that sequences of these combinations occur as each day passes.

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

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

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

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

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

**Inference**

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

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

It may not be immediately obvious, but every sensory experience you have ever had has been novel, yet you easily find familiar patterns in this novel input. For example, you can understand the word “breakfast” spoken by almost anyone, no matter whether they are old or young, male or female, are speaking quickly or slowly, or have a strong accent. Even if you had the same person say the same word “breakfast” a hundred times, the sound would never stimulate your cochleae (auditory receptors) in exactly the same way twice.

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

**Prediction**

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

Following are some key properties of HTM prediction.

**1) Prediction is continuous.**

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

**2) Prediction occurs in every region at every level of the hierarchy.**

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

**3) Predictions are context sensitive.**

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

**4) Prediction leads to stability.**

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

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

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

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

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

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

**Behavior**

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

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

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

**Progress toward the implementation of HTM**

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

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

**Chapter 2: HTM Cortical Learning Algorithms**

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

**Terminology**

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

**Cell states**

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

**Dendrite segments**

HTM cells have a relatively realistic (and therefore complex) dendrite model. In theory each HTM cell has one proximal dendrite segment and a dozen or two distal dendrite segments. The proximal dendrite segment receives feed-forward input and the distal dendrite segments receive lateral input from nearby cells. A class of inhibitory cells forces all the cells in a column to respond to similar feed-forward input. To simplify, we removed the proximal dendrite segment from each cell and replaced it with a single shared dendrite segment per column of cells. The spatial pooler function (described below) operates on the shared dendrite segment, at the level of columns. The temporal pooler function operates on distal dendrite segments, at the level of individual cells within columns. This simplification achieves the same functionality, though in biology there is no equivalent to a dendrite segment attached to a column.

**Synapses**

HTM synapses have binary weights. Biological synapses have varying weights but they are also partially stochastic, suggesting a biological neuron cannot rely on precise synaptic weights. The use of distributed representations in HTMs plus our model of dendrite operation allows us to assign binary weights to HTM synapses with no ill effect. To model the forming and un-forming of synapses we use two additional concepts from neuroscience that you may not be familiar with. One is the concept of “potential synapses”. This represents all the axons that pass close enough to a dendrite segment that they could potentially form a synapse. The second is called “permanence”. This is a scalar value assigned to each potential synapse. The permanence of a synapse represents a range of connectedness between an axon and a dendrite. Biologically, the range would go from completely unconnected, to starting to form a synapse but not connected yet, to a minimally connected synapse, to a large fully connected synapse. The permanence of a synapse is a scalar value ranging from 0.0 to 1.0. Learning involves incrementing and decrementing a synapse’s permanence. When a synapse’s permanence is above a threshold, it is connected with a weight of “1”. When it is below the threshold, it is unconnected with a weight of “0”.

**Overview**

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

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

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

We will discuss each of these steps in more detail.

**1) Form a sparse distributed representation of the input**

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

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

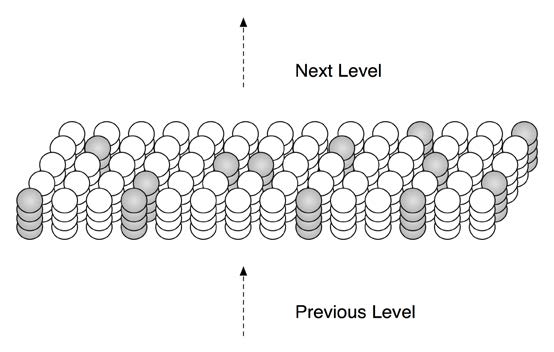


Figure 2.1: An HTM region consists of columns of cells. Only a small portion of a region is shown. Each column of cells receives activation from a unique subset of the input. Columns with the strongest activation inhibit columns with weaker activation. The result is a sparse distributed representation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

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

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

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

The next function performed by a region is to convert the columnar representation of the input into a new representation that includes state, or context, from the past. The new representation is formed by activating a subset of the cells within each column, typically only one cell per column (Figure 2.2).

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

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

Each column in an HTM region consists of multiple cells. 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. A specific example might help. Say every column has 4 cells and the representation of every input consists of 100 active columns. If only one cell per column is active at a time, we have 4100 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 4100 possible patterns will overlap by about 25 cells. Thus two representations of a particular input in different contexts will have about 25 cells in common and 75 cells that are different, making them easily distinguishable.

The general rule used by an HTM region is the following. When a column becomes active, it 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. You can think of it this way, if an input pattern is expected then the system confirms that expectation by activating only 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 “the input occurred unexpectedly so 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.

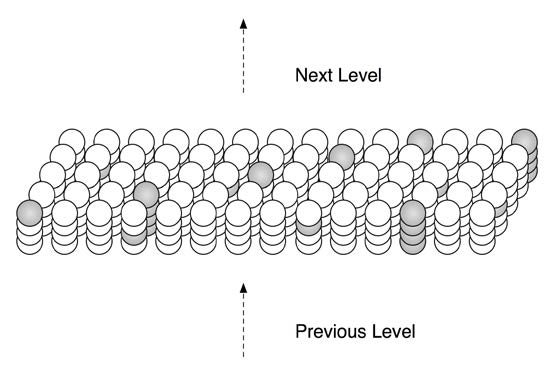


Figure 2.2: By activating a subset of cells in each column, an HTM region can represent the same input in many different contexts. Columns only activate predicted cells. Columns with no predicted cells activate all the cells in the column. The figure shows some columns with one cell active and some columns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

As mentioned in the terminology section above, HTM cells can be in one of three states. If a cell is active due to feed-forward input we just use the term “active”. If the cell is active due to lateral connections to other nearby cells we say it is in the “predictive state” (Figure 2.3).

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

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

When a region makes a prediction it activates (into the predictive state) all the cells that will likely become active due to future feed-forward input. Because representations in a region are sparse, multiple predictions can be made at the same time. For example if 2% of the columns are active due to an input, you could expect that ten different predictions could be made resulting in 20% of the columns having a predicted cell. Or, twenty different predictions could be made resulting in 40% of the columns having a predicted cell. If each column had four cells, with one active at a time, then 10% of the cells would be in the predictive state.

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

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

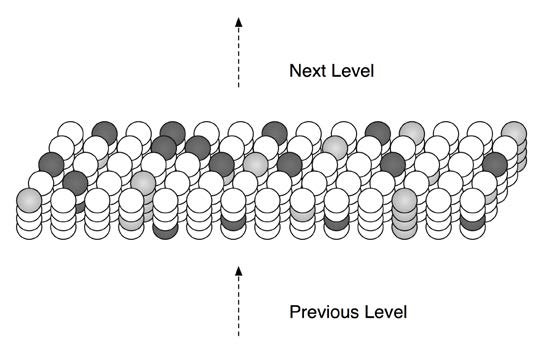


Figure 2.3: At any point in time, some cells in an HTM region will be active due to feed-forward input (shown in light gray). Other cells that receive lateral input from active cells will be in a predictive state (shown in dark gray).

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

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

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

We use the term “temporal pooler” to describe the two steps of adding context to the representation and predicting. By creating slowly changing outputs for sequences of patterns, we are in essence “pooling” together different patterns that follow each other in time. Now we will go into another level of detail. We start with concepts that are shared by the spatial pooler and temporal pooler. Then we discuss concepts and details unique to the spatial pooler followed by concepts and details unique to the temporal pooler.

**Shared concepts**

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

**Binary weights**

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

**Permanence**

Synapses are forming and unforming constantly during learning. As mentioned before, we assign a scalar value to each synapse (0.0 to 1.0) to indicate how permanent the connection is. When a connection is reinforced, its permanence is increased. Under other conditions, the permanence is decreased. When the permanence is above a threshold (e.g. 0.2), the synapse is considered to be established. If the permanence is below the threshold, the synapse will have no effect.

**Dendrite segments**

Synapses connect to dendrite segments. There are two types of dendrite segments, 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 region. Every cell has several distal dendrite segments. If the sum of the active synapses on a distal segment exceeds a threshold, then the associated cell becomes active in a 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**

As mentioned earlier, each dendrite segment has a list of potential synapses. All the potential synapses are given a permanence value and may become functional synapses if their permanence values exceed a threshold.

**Learning**

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

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

**Spatial pooler concepts**

The most fundamental function of the spatial pooler is to convert a region’s input into a sparse pattern. This function is important because the mechanism used to learn sequences and make predictions requires starting with sparse distributed patterns. There are several overlapping goals for the spatial pooler, which determine how the spatial pooler operates and learns.

**1) Use all columns**

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

**2) Maintain desired density**

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

**3) Avoid trivial patterns**

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

**4) Avoid extra connections**

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

**5) Self adjusting receptive fields**

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

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

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

**Spatial pooler details**

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

1. Start with an input consisting of a fixed number of bits. These bits might represent sensory data or they might come from another region lower in the hierarchy.
2. Assign a fixed number of columns to the region receiving this input. Each column has an associated dendrite segment. Each dendrite segment has a set of potential synapses representing a subset of the input bits. Each potential synapse has a permanence value. Based on their permanence values, some of the potential synapses will be valid.
3. For any given input, determine how many valid synapses on each column are connected to active input bits.
4. The number of active synapses is multiplied by a “boosting” factor which is dynamically determined by how often a column is active relative to its neighbors.
5. The columns with the highest activations after boosting disable all but a fixed percentage of the columns within an inhibition radius. The inhibition radius is itself dynamically determined by the spread (or “fan-out”) of input bits. There is now a sparse set of active columns.
6. For each of the active columns, we adjust the permanence values of all the potential synapses. The permanence values of synapses aligned with active input bits are increased. The permanence values of synapses aligned with inactive input bits are decreased. The changes made to permanence values may change some synapses from being valid to not valid, and vice-versa.

**Temporal pooler concepts**

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

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

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

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

Every cell participates in many different distributed patterns and in many different sequences. A particular cell might be part of dozens or hundreds of temporal transitions. Therefore every cell has several dendrite segments, not just one. Ideally a cell would have one dendrite segment for each pattern of activity it wants to recognize. Practically though, a dendrite segment can learn connections for several completely different patterns and still work well. For example, one segment might learn 20 connections for each of 4 different patterns, for a total of 80 connections. We 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.

**Temporal pooler details**

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

1. For each active column, check for cells in the column that are in a predictive state, and activate them. If no cells are in a predictive state, activate all the cells in the column. The resulting set of active cells is the representation of the input in the context of prior input.
2. For every dendrite segment on every cell in the region, count how many established synapses are connected to active cells. If the number exceeds a threshold, that dendrite segment is marked as active. Cells with active dendrite segments are put in the predictive state unless they are already active due to feed-forward input. Cells with no active dendrites and not active due to bottom-up input become or remain inactive. The collection of cells now in the predictive state is the prediction of the region.
3. When a dendrite segment becomes active, modify the permanence values of all the synapses associated with the segment. For every potential synapse on the active dendrite segment, increase the permanence of those synapses that are connected to active cells and decrement the permanence of those synapses connected to inactive cells. These changes to synapse permanence are marked as temporary. This modifies the synapses on segments that are already trained sufficiently to make the segment active, and thus lead to a prediction. However, we always want to extend predictions further back in time if possible. Thus, we pick a second dendrite segment on the same cell to train. For the second segment we choose the one that best matches the state of the system in the previous time step. For this segment, using the state of the system in the previous time step, increase the permanence of those synapses that are connected to active cells and decrement the permanence of those synapses connected to inactive cells. These changes to synapse permanence are marked as temporary.
4. Whenever a cell switches from being inactive to active due to feed-forward input, we traverse each potential synapse associated with the cell and remove any temporary marks. Thus we update the permanence of synapses only if they correctly predicted the feed-forward activation of the cell.
5. When a cell switches from either active state to inactive, undo any permanence changes marked as temporary for each potential synapse on this cell. We don’t want to strengthen the permanence of synapses that incorrectly predicted the feed-forward activation of a cell. Note that only cells that are active due to feed-forward input propagate activity within the region, otherwise predictions would lead to further predictions. But all the active cells (feed-forward and predictive) form the output of a region and propagate to the next region in the hierarchy.

**First order versus variable order sequences and prediction**

There is one more major topic to discuss before we end our discussion on the spatial and temporal poolers. It may not be of interest to all readers and it is not needed to understand Chapters 3 and 4. What is the effect of having more or fewer cells per column? Specifically, what happens if we have only one cell per column?

In the example used earlier, we showed that a representation 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 region 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 region 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 region is a variable order memory.

If we increase to five cells per column, the available number of encodings of any particular input in our example would increase to 5100, a huge increase over 4100. But both these numbers are so large that for many practical problems the increase in capacity might not be useful.

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

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

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

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

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

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

If we now do the same experiment on an HTM region with multiple cells per column, we find that the cells behave like “directionally-tuned complex cells” in the neocortex. The predictive output of a cell will be active for a line moving to the left or a line moving to the right, but not both.

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

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

**Chapter 3: Spatial Pooling Implementation and Pseudocode**

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

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

Phase 1: compute the overlap with the current input for each column

Phase 2: compute the winning columns after inhibition

Phase 3: update synapse permanence and internal variables

Although spatial pooler learning is inherently online, you can turn off learning by simply skipping Phase 3. The rest of the chapter contains the pseudocode for each of the three steps. The various data structures and supporting routines used in the code are defined at the end.

**Initialization**

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

**Phase 1: Overlap**

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

1 **for** column in columns

2

3 column.overlap = 0

4

5 **for** synapse in column.connectedSynapses

6 column.overlap += synapse.inputBit(timestep)

7

8 **if** column.overlap < minOverlap then

9 column.overlap = 0

10 **else**

11 column.overlap \*= column.boost

**Phase 2: Inhibition**

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

1 **for** column in columns

2 minLocalActivity = kthScore(column.neighbors, desiredLocalActivity)

3

4 **if** column.overlap > 0 and column.overlap â‰¥ minLocalActivity then

5 region.activeColumns(timestep).append(column)

**Phase 3: Learning**

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

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

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

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

1 **for** column in region.activeColumns(timestep)

2 **for** synapse in column.potentialSynapses

3 **if** synapse.isActive then

4 synapse.permanence += permanenceIncrement

5 synapse.permanence = min(1.0, synapse.permanence)

6 **else**

7 synapse.permanence -= permanenceDecrement

8 synapse.permanence = max(0.0, synapse.permanence)

9

10 **for** column in region.columns

11 column.minDutyCycle = 0.01 \* maxDutyCycle(column.neighbors)

12 column.updateActiveDutyCycle()

13 column.boost = boost(column.activeDutyCycle, column.minDutyCycle)

14

15 column.updateOverlapDutyCycle()

16 **if** column.overlapDutyCycle < column.minDutyCycle then

17 column.increasePermanences(0.1\*connectedPerm)

18

19 inhibitionRadius = averageReceptiveFieldSize()

**Supporting data structures and routines**

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

| **Variable** | **Description** |
| --- | --- |
| region.columns | List of all columns. |
| inputBit(timestep,bit) | The input to this level at timestep, 1 if bit bit is on. |
| column.overlap | The spatial pooler overlap of column with a particular input pattern. |
| activeColumns(timestep) | List of column indices that are winners due to bottom-up input. |
| desiredLocalActivity | A parameter controlling the number of columns that will be winners after the inhibition step. |
| inhibitionRadius | Average connected receptive field size of the columns. |
| column.neighbors | A list of all the columns that are within inhibitionRadius of column. |
| minOverlap | A minimum number of inputs that must be active for a column to be considered during the inhibition step. |
| column.boost | The boost value for column as computed during learning - used to increase the overlap value for inactive columns. |
| synapse | A data structure representing a synapse - contains a permanence value and the source input index. |
| connectedPerm | If the permanence value for a synapse is greater than this value, it is said to be connected. |
| column.potentialSynapses | The list of potential synapses and their permanence values. |
| column.connectedSynapses | A subset of column.potentialSynapses where the permanence value is greater than connectedPerm. These are the bottom-up inputs that are currently connected to column. |
| permanenceIncrement | Amount permanence values of synapses are incremented during learning. |
| permanenceDecrement | Amount permanence values of synapses are decremented during learning. |
| column.activeDutyCycle | A sliding average representing how often column has been active after inhibition (e.g. over the last 1000 iterations). |
| column.overlapDutyCycle | A sliding average representing how often column has had significant overlap (i.e. greater than minOverlap) with its inputs (e.g. over the last 1000 iterations). |
| column.minDutyCycle | A variable representing the minimum desired firing rate for a cell. If a cell’s firing rate falls below this value, it will be boosted. This value is calculated as 1% of the maximum firing rate of its neighbors. |

The following supporting routines are used in the above code.

| **Function** | **Description** |
| --- | --- |
| kthScore(columns, k) | Given the list of columns, return the k’th highest overlap value. |
| column.updateActiveDutyCycle() | Computes a moving average of how often column c has been active after inhibition. |
| column.updateOverlapDutyCycle() | Computes a moving average of how often column has overlap greater than minOverlap. |
| averageReceptiveFieldSize() | The radius of the average connected receptive field size of all the columns. The connected receptive field size of a column includes only the connected synapses (those with permanence values >= connectedPerm). This is used to determine the extent of lateral inhibition between columns. |
| maxDutyCycle(cols) | Returns the maximum active duty cycle of the columns in the given list of columns. |
| column.increasePermanences(scale) | Increase the permanence value of every synapse in column by a scale factor scale. |
| column.boost | Returns the boost value of a column. The boost value is a scalar >= 1. If column.activeDutyCycle is above column.minDutyCycle, the boost value is 1. The boost increases linearly once the column’s activeDutyCycle starts falling below its minDutyCycle. |

**Chapter 4: Temporal Pooling Implementation and Pseudocode**

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

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

Phase 1: compute the active state, activeState(t), for each cell

Phase 2: compute the predicted state, predictiveState(t), for each cell

Phase 3: update synapses

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

**Temporal pooler pseudocode: inference alone**

**Phase 1**

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

1 **for** c in activeColumns(t)

2 buPredicted = false

3 **for** i = 0 to cellsPerColumn - 1

4 **if** predictiveState(c, i, t-1) == true then

5 s = getActiveSegment(c, i, t-1, activeState)

6 **if** s.sequenceSegment == true then

7 buPredicted = true

8 activeState(c, i, t) = 1

9

10 **if** buPredicted == false then

11 **for** i = 0 to cellsPerColumn - 1

12 activeState(c, i, t) = 1

**Phase 2**

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

1 **for** c, i in cells

2 **for** s in segments(c, i)

3 **if** segmentActive(c, i, s, t) then

4 predictiveState(c, i, t) = 1

**Temporal pooler pseudocode: combined inference and learning**

**Phase 1**

The first phase calculates the activeState for each cell that is in a winning column. For those columns, the code further selects one cell per column as the learning cell (learnState). The logic is as follows: if the bottom-up input was predicted by any cell (i.e. its predictiveState output was 1 due to a sequence segment), then those cells become active (lines 23-27). If that segment became active from cells chosen with learnState on, this cell is selected as the learning cell (lines 28-30). If the bottom-up input was not predicted, then all cells in the become active (lines 32-34). In addition, the best matching cell is chosen as the learning cell (lines 36-41) and a new segment is added to that cell.

1 **for** c in activeColumns(t)

2 buPredicted = false

3 lcChosen = false

4 **for** i = 0 to cellsPerColumn - 1

5 **if** predictiveState(c, i, t-1) == true then

6 s = getActiveSegment(c, i, t-1, activeState)

7 **if** s.sequenceSegment == true then

8 buPredicted = true

9 activeState(c, i, t) = 1

10 **if** segmentActive(s, t-1, learnState) then

11 lcChosen = true

12 learnState(c, i, t) = 1

13

14 **if** buPredicted == false then

15 **for** i = 0 to cellsPerColumn - 1

16 activeState(c, i, t) = 1

17

18 **if** lcChosen == false then

19 I,s = getBestMatchingCell(c, t-1)

20 learnState(c, i, t) = 1

21 sUpdate = getSegmentActiveSynapses (c, i, s, t-1, true)

22 sUpdate.sequenceSegment = true

23 segmentUpdateList.add(sUpdate)

**Phase 2**

The second phase calculates the predictive state for each cell. A cell will turn on its predictive state output if one of its segments becomes active, i.e. if enough of its lateral inputs are currently active due to feed-forward input. In this case, the cell queues up the following changes: a) reinforcement of the currently active segment (lines 47-48), and b) reinforcement of a segment that could have predicted this activation, i.e. a segment that has a (potentially weak) match to activity during the previous time step (lines 50-53).

1 **for** c, i in cells

2 **for** s in segments(c, i)

3 **if** segmentActive(s, t, activeState) then

4 predictiveState(c, i, t) = 1

5

6 activeUpdate = getSegmentActiveSynapses (c, i, s, t, false)

7 segmentUpdateList.add(activeUpdate)

8

9 predSegment = getBestMatchingSegment(c, i, t-1)

10 predUpdate = getSegmentActiveSynapses(c, i, predSegment, t-1, true)

11 segmentUpdateList.add(predUpdate)

**Phase 3**

The third and last phase actually carries out learning. In this phase segment updates that have been queued up are actually implemented once we get feed-forward input and the cell is chosen as a learning cell (lines 56-57). Otherwise, if the cell ever stops predicting for any reason, we negatively reinforce the segments (lines 58-60).

1 **for** c, i in cells

2 **if** learnState(s, i, t) == 1 then

3 adaptSegments (segmentUpdateList(c, i), true)

4 segmentUpdateList(c, i).delete()

5 **else** **if** predictiveState(c, i, t) == 0

6 and predictiveState(c, i, t-1)==1 then

7 adaptSegments (segmentUpdateList(c,i), false)

8 segmentUpdateList(c, i).delete()

**Implementation details and terminology**

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

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

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

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

| **Variable** | **Description** |
| --- | --- |
| cell(c,i) | A list of all cells, indexed by i and c. |
| cellsPerColumn | Number of cells in each column. |
| activeColumns(t) | List of column indices that are winners due to bottom-up input (this is the output of the spatial pooler). |
| activeState(c, i, t) | A boolean vector with one number per cell. It represents the active state of the column c cell i at time t given the current feed-forward input and the past temporal context. activeState(c, i, t) is the contribution from column c cell i at time t. If 1, the cell has current feed-forward input as well as an appropriate temporal context. |
| predictiveState(c, i, t) | A boolean vector with one number per cell. It represents the prediction of the column c cell i at time t, given the bottom-up activity of other columns and the past temporal context. predictiveState(c, i, t) is the contribution of column c cell i at time t. If 1, the cell is predicting feed-forward input in the current temporal context. |
| learnState(c, i, t) | A boolean indicating whether cell i in column c is chosen as the cell to learn on. |
| activationThreshold | Activation threshold for a segment. If the number of active connected synapses in a segment is greater than activationThreshold, the segment is said to be active. |
| learningRadius | The area around a temporal pooler cell from which it can get lateral connections. |
| initialPerm | Initial permanence value for a synapse. |
| connectedPerm | If the permanence value for a synapse is greater than this value, it is said to be connected. |
| minThreshold | Minimum segment activity for learning. |
| newSynapseCount | The maximum number of synapses added to a segment during learning. |
| permanenceInc | Amount permanence values of synapses are incremented when activity-based learning occurs. |
| permanenceDec | Amount permanence values of synapses are decremented when activity-based learning occurs. |
| segmentUpdate | Data structure holding three pieces of information required to update a given segment: a) segment index (-1 if it’s a new segment), b) a list of existing active synapses, and c) a flag indicating whether this segment should be marked as a sequence segment (defaults to false). |
| segmentUpdateList | A list of segmentUpdate structures. segmentUpdateList(c,i) is the list of changes for cell i in column c. |

The following supporting routines are used in the above code:

| **Function** | **Description** |
| --- | --- |
| segmentActive(s, t, state) | This routine returns true if the number of connected synapses on segment s that are active due to the given state at time t is greater than activationThreshold. The parameter state can be activeState, or learnState. |
| getActiveSegment(c, i, t, state) | For the given column c cell i, return a segment index such that segmentActive(s,t, state) is true. If multiple segments are active, sequence segments are given preference. Otherwise, segments with most activity are given preference. |
| getBestMatchingSegment(c, i, t) | For the given column c cell i at time t, find the segment with the largest number of active synapses. This routine is aggressive in finding the best match. The permanence value of synapses is allowed to be below connectedPerm. The number of active synapses is allowed to be below activationThreshold, but must be above minThreshold. The routine returns the segment index. If no segments are found, then an index of -1 is returned. |
| getBestMatchingCell(c) | For the given column, return the cell with the best matching segment (as defined above). If no cell has a matching segment, then return the cell with the fewest number of segments. |
| getSegmentActiveSynapses(c, i, t, s, newSynapses= false) | Return a segmentUpdate data structure containing a list of proposed changes to segment s. Let activeSynapses be the list of active synapses where the originating cells have their activeState output = 1 at time step t. (This list is empty if s = -1 since the segment doesn’t exist.) newSynapses is an optional argument that defaults to false. If newSynapses is true, then newSynapseCount - count(activeSynapses) synapses are added to activeSynapses. These synapses are randomly chosen from the set of cells that have learnState output = 1 at time step t. |
| adaptSegments(segmentList, positiveReinforcement) | This function iterates through a list of segmentUpdate’s and reinforces each segment. For each segmentUpdate element, the following changes are performed. If positiveReinforcement is true then synapses on the active list get their permanence counts incremented by permanenceInc. All other synapses get their permanence counts decremented by permanenceDec. If positiveReinforcement is false, then synapses on the active list get their permanence counts decremented by permanenceDec. After this step, any synapses in segmentUpdate that do yet exist get added with a permanence count of initialPerm. |

**Appendix A: A Comparison between Biological Neurons and HTM Cells**

The image above shows a picture of a biological neuron on the left, a simple artificial neuron in the middle, and an HTM neuron or “cell” on the right. The purpose of this appendix is to provide a better understanding of HTM cells and how they work by comparing them to real neurons and simpler artificial neurons.

Real neurons are tremendously complicated and varied. We will focus on the most general principles and only those that apply to our model. Although we ignore many details of real neurons, the cells used in the HTM cortical learning algorithms are far more realistic than the artificial neurons used in most neural networks. All the elements included in HTM cells are necessary for the operation of an HTM region.

**Biological neurons**

Neurons are the information carrying cells in the brain. The image on the left above is of a typical excitatory neuron. The visual appearance of a neuron is dominated by the branching dendrites. All the excitatory inputs to a neuron are via synapses aligned along the dendrites. In recent years our knowledge of neurons has advanced considerably. The biggest change has been in realizing that the dendrites of a neuron are not just conduits to bring inputs to the cell body. We now know the dendrites are complex non-linear processing elements in themselves. The HTM cortical learning algorithms take advantage of these non-linear properties.

Neurons have several parts.

**Cell body**

The cell body is the small volume in the center of the neuron. The output of the cell, the axon, originates at the cell body. The inputs to the cell are the synapses aligned along the dendrites which feed to the cell body.

**Proximal Dendrites**

The dendrite branches closest to the cell body are called proximal dendrites. In the diagram some of the proximal dendrites are marked with green lines.

Multiple active synapses on proximal dendrites have a roughly linear additive effect at the cell body. Five active synapses will lead to roughly five times the depolarization at the cell body compared to one active synapse. In contrast, if a single synapse is activated repeatedly by a quick succession of action potentials, the second, third, and subsequent action potentials have much less effect at the cell body, than the first.

Therefore, we can say that inputs to the proximal dendrites sum linearly at the cell body, and that rapid spikes arriving at a single synapse will have only a slightly larger effect than a single spike.

The feed-forward connections to a region of neocortex preferentially connect to the proximal dendrites. This has been reported at least for layer 4 neurons, the primary input layer of neurons in each region.

**Distal Dendrites**

The dendrite branches farther from the cell body are called distal dendrites. In the diagram some of the distal dendrites are marked with blue lines.

Distal dendrites are thinner than proximal dendrites. They connect to other dendrites at branches in the dendritic tree and do not connect directly to the cell body. These differences give distal dendrites unique electrical and chemical properties. When a single synapse is activated on a distal dendrite, it has a minimal effect at the cell body. The depolarization that occurs locally to the synapse weakens by the time it reaches the cell body. For many years this was viewed as a mystery. It seemed the distal synapses, which are the majority of synapses on a neuron, couldn’t do much.

We now know that sections of distal dendrites act as semi-independent processing regions. If enough synapses become active at the same time within a short distance along the dendrite, they can generate a dendritic spike that can travel to the cell body with a large effect. For example, twenty active synapses within 40 um of each other will generate a dendritic spike.

Therefore, we can say that the distal dendrites act like a set of threshold coincidence detectors.

The synapses formed on distal dendrites are predominantly from other cells nearby in the region.

The image shows a large dendrite branch extending upwards which is called the apical dendrite. One theory says that this structure allows the neuron to locate several distal dendrites in an area where they can more easily make connections to passing axons. In this interpretation, the apical dendrite acts as an extension of the cell.

**Synapses**

A typical neuron might have several thousand synapses. The large majority (perhaps 90%) of these will be on distal dendrites, and the rest will be on proximal dendrites.

For many years it was assumed that learning involved strengthening and weakening the effect or “weight” of synapses. Although this effect has been observed, each synapse is somewhat stochastic. When activated, it will not reliably release a neurotransmitter. Therefore the algorithms used by the brain cannot depend on precision or fidelity of individual synapse weights. Further, we now know that entire synapses form and un-form rapidly. This flexibility represents a powerful form of learning and better explains the rapid acquisition of knowledge. A synapse can only form if an axon and a dendrite are within a certain distance, leading to the concept of “potential” synapses. With these assumptions, learning occurs largely by forming valid synapses from potential synapses.

**Neuron Output**

The output of a neuron is a spike, or “action potential”, which propagates along the axon. The axon leaves the cell body and almost always splits in two. One branch travels horizontally making many connections with other cells nearby. The other branch projects to other layers of cells or elsewhere in the brain. In the image of the neuron above, the axon was not visible. We added a line and two arrows to represent that axon.

Although the actual output of a neuron is always a spike, there are different views on how to interpret this. The predominant view (especially in regards to the neocortex) is that the rate of spikes is what matters. Therefore the output of a cell can be viewed as a scalar value.

Some neurons also exhibit a “bursting” behavior, a short and fast series of a few spikes that are different than the regular spiking pattern.

The above description of a neuron is intended to give a brief introduction to neurons. It focuses on attributes that correspond to features of HTM cells and leaves out many details. Not all the features just described are universally accepted. We include them because they are necessary for our models. What is known about neurons could easily fill several books, and active research on neurons continues today.

**Simple artificial neurons**

The middle image at the beginning of this Appendix shows a neuron-like element used in many classic artificial neural network models. These artificial neurons have a set of synapses each with a weight. Each synapse receives a scalar activation, which is multiplied by the synapse weight. The output of all the synapses is summed in a non-linear fashion to produce an output of the artificial neuron. Learning occurs by adjusting the weights of the synapses and perhaps the non-linear function.

This type of artificial neuron, and variations of it, has proven useful in many applications as a valuable computational tool. However, it doesn’t capture much of the complexity and processing power of biological neurons. If we want to understand and model how an ensemble of real neurons works in the brain we need a more sophisticated neuron model.

**HTM cells**

In our illustration, the image on the right depicts a cell used in the HTM cortical learning algorithms. An HTM cell captures many of the important capabilities of real neurons but also makes several simplifications.

**Proximal Dendrite**

Each HTM cell has a single proximal dendrite. All feed-forward inputs to the cell are made via synapses (shown as green dots). The activity of synapses is linearly summed to produce a feed-forward activation for the cell.

We require that all cells in a column have the same feed-forward response. In real neurons this would likely be done by a type of inhibitory cell. In HTMs we simply force all the cells in a column to share a single proximal dendrite.

To avoid having cells that never win in the competition with neighboring cells, an HTM cell will boost its feed-forward activation if it is not winning enough relative to its neighbors. Thus there is a constant competition between cells. Again, in an HTM we model this as a competition between columns, not cells. This competition is not illustrated in the diagram.

Finally, the proximal dendrite has an associated set of potential synapses which is a subset of all the inputs to a region. As the cell learns, it increases or decreases the “permanence” value of all the potential synapses on the proximal dendrite. Only those potential synapses that are above a threshold are valid.

As mentioned earlier, the concept of potential synapses comes from biology where it refers to axons and dendrites that are close enough to form a synapse. We extend this concept to a larger set of potential connections for an HTM cell. Dendrites and axons on biological neurons can grow and retract as learning occurs and therefore the set of potential synapses changes with growth. By making the set of potential synapses on an HTM cell large, we roughly achieve the same result as axon and dendrite growth. The set of potential synapses is not shown.

The combination of competition between columns, learning from a set of potential synapses, and boosting underutilized columns gives a region of HTM neurons a powerful plasticity also seen in brains. An HTM region will automatically adjust what each column represents (via changes to the synapses on the proximal dendrites) if the input changes, or the number of columns increases or decreases.

**Distal Dendrites**

Each HTM cell maintains a list of distal dendrite segments. Each segment acts like a threshold detector. If the number of active synapses on any segment (shown as blue dots on the earlier diagram) is above a threshold, the segment becomes active, and the associated cell enters the predictive state. The predictive state of a cell is the OR of the activations of its segments.

A dendrite segment remembers the state of the region by forming connections to cells that were active together at a point in time. The segment remembers a state that precedes the cell becoming active due to feed-forward input. Thus the segment is looking for a state that predicts that its cell will become active. A typical threshold for a dendrite segment is 15. If 15 valid synapses on a segment are active at once, the dendrite becomes active. There might be hundreds or thousands of cells active nearby, but connecting to only 15 is sufficient to recognize the larger pattern.

Each distal dendrite segment also has an associated set of potential synapses. The set of potential synapses is a subset of all the cells in a region. As the segment learns, it increases or decreases the permanence value of all its potential synapses. Only those potential synapses that are above a threshold are valid.

In one implementation, we use a fixed number of dendrite segments per cell. In another implementation, we add and delete segments while training. Both methods can work. If we have a fixed number of dendrite segments per cell, it is possible to store several different sets of synapses on the same segment. For example, say we have 20 valid synapses on a segment and a threshold of 15. (In general we want the threshold to be less than the number of synapses to improve noise immunity.) The segment can now recognize one particular state of the cells nearby. What would happen if we added another 20 synapses to the same segment representing an entirely different state of cells nearby? It introduces the possibility of error because the segment could add 8 active synapses from one pattern and 7 active synapses from the other and become active incorrectly. We have found experimentally that up to 20 different patterns can be stored on one segment before errors occur. Therefore an HTM cell with a dozen dendrite segments can participate in many different predictions.

**Synapses**

Synapses on an HTM cell have a binary weight. There is nothing in the HTM model that precludes scalar synapse weights, but due to the use of sparse distributed patterns we have not yet had a need to use scalar weights.

However, synapses on an HTM cell have a scalar value called “permanence” which is adjusted during learning. A 0.0 permanence value represents a potential synapse which is not valid and has not progressed at all towards becoming a valid synapse. A permanence value above a threshold (typically 0.2) represents a synapse that has just connected but could easily be un-connected. A high permanence value, for example 0.9, represents a synapse that is connected and cannot easily be un-connected.

The number of valid synapses on the proximal and distal dendrite segments of an HTM cell is not fixed. It changes as the cell is exposed to patterns. For example, the number of valid synapses on the distal dendrites is dependent on the temporal structure of the data. If there are no persistent temporal patterns in the input to the region, then all the synapses on distal segments would have low permanence values and very few synapses would be valid. If there is a lot of temporal structure in the input stream, then we will find many valid synapses with high permanence.

**Cell Output**

An HTM cell has two different binary outputs: 1) the cell is active due to feed-forward input (via the proximal dendrite), and 2) the cell is active due to lateral connections (via the distal dendrite segments). The former is called the “active state” and the latter is called the “predictive state”.

In the earlier diagram, the two outputs are represented by the two lines exiting the square cell body. The left line is the feed-forward active state, while the right line is the predictive state.

Only the feed-forward active state is connected to other cells in the region, ensuring that predictions are always based on the current input (plus context). We don’t want to make predictions based on predictions. If we did, almost all the cells in the region would be in the predictive state after a few iterations.

The output of the region is a vector representing the state of all the cells. This vector becomes the input to the next region of the hierarchy if there is one. This output is the OR of the active and predictive states. By combining both active and predictive states, the output of our region will be more stable (slower changing) than the input. Such stability is an important property of inference in a region.

**Suggested reading**

We are often asked to suggest reading materials to learn more about neuroscience. The field of neuroscience is so large that a general introduction requires looking at many different sources. New findings are published in academic journals which are both hard to read and hard to get access to if you don’t have a university affiliation.

Here are two readily available books that a dedicated reader might want to look at which are relevant to the topics in this appendix.

Stuart, Greg, Spruston, Nelson, Häusser, Michael, Dendrites, second edition (New York: Oxford University Press, 2008) This book is a good source on everything about dendrites. Chapter 16 discusses the non-linear properties of dendrite segments used in the HTM cortical learning algorithms. It is written by Bartlett Mel who has done much of the thinking in this field.

Mountcastle, Vernon B. Perceptual Neuroscience: The Cerebral Cortex (Cambridge, Mass.: Harvard University Press, 1998) This book is a good introduction to everything about the neocortex. Several of the chapters discuss cell types and their connections. You can get a good sense of cortical neurons and their connections, although it is too old to cover the latest knowledge of dendrite properties.

**Appendix B: A Comparison of Layers in the Neocortex and an HTM Region**

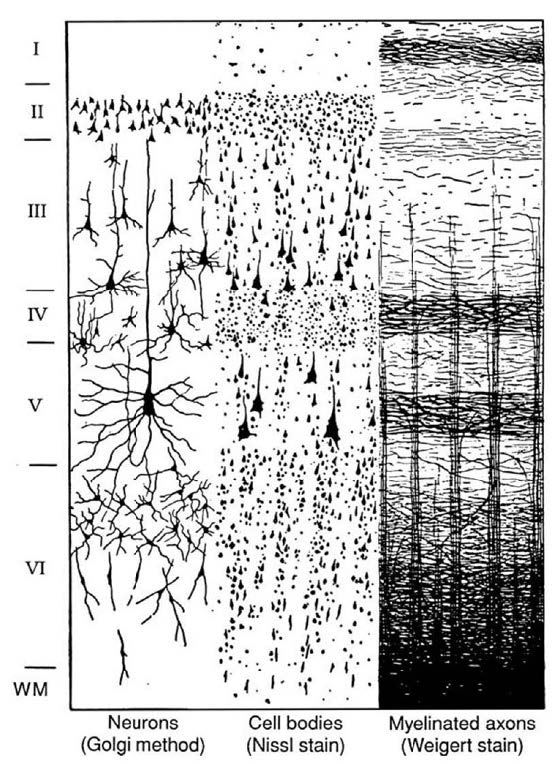
This appendix describes the relationship between an HTM region and a region of the biological neocortex.

Specifically, the appendix covers how the HTM cortical learning algorithm, with its columns and cells, relates to the layered and columnar architecture of the neocortex. Many people are confused by the concept of “layers” in the neocortex and how it relates to an HTM layer. Hopefully this appendix will resolve this confusion as well as provide more insight into the biology underlying the HTM cortical learning algorithm.

**Circuitry of the neocortex**

The human neocortex is a sheet of neural tissue approximately 1,000 cm2 in area and 2mm thick. To visualize this sheet, think of a cloth dinner napkin, which is a reasonable approximation of the area and thickness of the neocortex. The neocortex is divided into dozens of functional regions, some related to vision, others to audition, and others to language, etc. Viewed under a microscope, the physical characteristics of the different regions look remarkably similar.

There are several organizing principles seen in each region throughout the neocortex.



**Layers**

The neocortex is generally said to have six layers. Five of the layers contain cells and one layer is mostly connections. The layers were discovered over one hundred years ago with the advent of staining techniques. The image above (from Cajal) shows a small slice of neocortex exposed using three different staining methods. The vertical axis spans the thickness of the neocortex, approximately 2mm. The left side of the image indicates the six layers. Layer 1, at the top, is the non-cellular level. The “WM” at the bottom indicates the beginning of the white matter, where axons from cells travel to other parts of the neocortex and other parts of the brain.

The right side of the image is a stain that shows only myelinated axons. (Myelination is a fatty sheath that covers some but not all axons.) In this part of the image you can see two of the main organizing principles of the neocortex, layers and columns. Most axons split in two immediately after leaving the body of the neuron. One branch will travel mostly horizontally and the other branch will travel mostly vertically. The horizontal branch makes a large number of connections to other cells in the same or nearby layer, thus the layers become visible in stains such as this. Bear in mind that this is a drawing of a slice of neocortex. Most of the axons are coming in and out of the plane of the image so the axons are longer than they appear in the image. It has been estimated that there are between 2 and 4 kilometers of axons and dendrites in every cubic millimeter of neocortex.

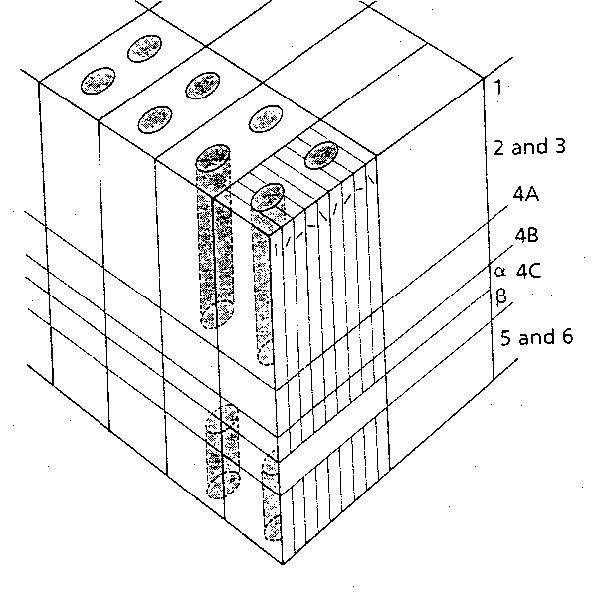
The middle section of the image is a stain that shows neuron bodies, but does not show any dendrites or axons. You can see that the size and density of the neurons also varies by layer. There is only a little indication of columns in this particular image. You might notice that there are some neurons in layer 1. The number of layer 1 neurons is so small that the layer is still referred to as a non-cellular layer. Neuro-scientists have estimated that there is somewhere around 100,000 neurons in a cubic millimeter of neocortex.

The left part of the image is a stain that shows the body, axons, and dendrites of just a few neurons. You can see that the size of the dendrite “arbors” varies significantly in cells in different layers. Also visible are some “apical dendrites” that rise from the cell body making connections in other layers. The presence and destination of apical dendrites is specific to each layer.

In short, the layered and columnar organization of the neocortex becomes evident when the neural tissue is stained and viewed under a microscope. Variations of layers in different regions There is variation in the thickness of the layers in different regions of the neocortex and some disagreement over the number of layers. The variations depend on what animal is being studied, what region is being looked at, and who is doing the looking. For example, in the image above, layer 2 and layer 3 look easily distinguished, but generally this is not the case. Some scientists report that they cannot distinguish the two layers in the regions they study, so often layer 2 and layer 3 are grouped together and called “layer 2/3”. Other scientists go the opposite direction, defining sub-layers such as 3A and 3B. Layer 4 is most well defined in those neocortical regions which are closest to the sensory organs. While in some animals (for example humans and monkeys), layer 4 in the first vision region is clearly subdivided. In other animals it is not subdivided. Layer 4 mostly disappears in regions hierarchically far from the sensory organs.

**Columns**

The second major organizing principle of the neocortex is columns. Some columnar organization is visible in stained images, but most of the evidence for columns is based on how cells respond to different inputs. When scientists use probes to see what makes neurons become active, they find that neurons that are vertically aligned, across different layers, respond to roughly the same input.



This drawing illustrates some of the response properties of cells in V1, the first cortical region to process information from the retina.

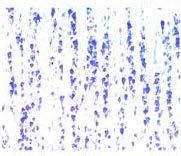
One of the first discoveries was that most cells in V1 respond to lines or edges at different orientations at specific areas of the retina. Cells that are vertically aligned in columns all respond to edges with the same orientation. If you look carefully, you will see that the drawing shows a set of small lines at different orientations arrayed across the top of the section. These lines indicate what line orientation cells at that location respond to. Cells that are vertically aligned (within the thin vertical stripes) respond to the lines of the same orientation.

There are several other columnar properties seen in V1, two of which are shown in the drawing. There are “ocular dominance columns” where cells respond to similar combinations of left and right eye influence. And there are “blobs” where cells are primarily color sensitive. The ocular dominance columns are the larger blocks in the diagram. Each ocular dominance column includes a set of orientation columns. The “blobs” are the dark ovals.

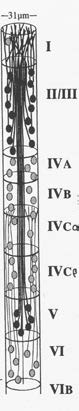
The general rule for neocortex is that several different response properties are overlaid on one another, such as orientation and ocular dominance. As you move horizontally across the cortical surface, the combination of response properties exhibited by cells changes. However, vertically aligned neurons share the same set of response properties. This vertical alignment is true in auditory, visual, and somatosensory areas. There is some debate amongst neuroscientists whether this is true everywhere in the neocortex but it appears to be true in most areas if not all.

**Mini-columns**

The smallest columnar structure in the neocortex is the mini-column. Mini-columns are about 30um in diameter and contain 80-100 neurons across all five cellular layers. The entire neocortex is composed of mini-columns. You can visualize them as tiny pieces of spaghetti stacked side by side. There are tiny gaps with few cells between the mini-columns sometimes making them visible in stained images.



A stained image that shows neuron cell bodies in part of a neocortical slice. The vertical structure of mini-columns is evident in this image.



A conceptual drawing of a mini-column (from Peters and Yilmez). In reality is skinnier than this. Note there are multiple neurons in each layer in the column. All the neurons in a mini-column will respond to similar inputs. For example, in the drawing of a section of V1 shown previously, a mini-column will contain cells that respond to lines of a particular orientation with a particular ocular dominance preference. The cells in an adjacent mini-column might respond to a slightly different line orientation or different ocular dominance preference.

Inhibitory neurons play an essential role is defining mini-columns. They are not visible in the image or drawing but inhibitory neurons send axons in a straight path between mini-columns partially giving them their physical separation. The inhibitory neurons are also believed to help force all the cells in the mini-column to respond to similar inputs.

The mini-column is the prototype for the column used in the HTM cortical learning algorithm. An exception to columnar responses There is a one exception to columnar responses that is relevant to the HTM cortical learning algorithms. Usually scientists find what a cell responds to by exposing an experimental animal to a simple stimulus. For example, they might show an animal a single line in a small part of the visual space to determine the response properties of cells in V1. When using simple inputs, researchers find that cells always will respond to the same input. However, if the simple input is embedded in a video of a natural scene, cells become more selective. A cell that reliably responds to an isolated vertical line will not always respond when the vertical line is embedded in a complex moving image of a natural scene.

In the HTM cortical learning algorithm, all HTM cells in a column share the same feed-forward response properties, but in a learned temporal sequence, only one of the cells in an HTM column becomes active. This mechanism is the means of representing variable order sequences and is analogous to the property just described for neurons. A simple input with no context will cause all the cells in a column to become active. The same input within a learned sequence will cause just one cell to become active. We are not suggesting that only one neuron within a mini-column will be active at once. The HTM cortical learning algorithm suggests that within a column, all the neurons within a layer would be active for an unanticipated input and a subset of the neurons would be active for an anticipated input.

**Why are there layers and columns?**

No one knows for certain why there are layers and why there are columns in the neocortex. HTM theory, however, proposes an answer. The HTM cortical learning algorithm shows that a layer of cells organized in columns can be a high capacity memory of variable order state transitions. Stated more simply, a layer of cells can learn a lot of sequences. Columns of cells that share the same feed-forward response are the key mechanism for learning variable-order transitions.

This hypothesis explains why columns are necessary, but what about the five layers? If a single cortical layer can learn sequences and make predictions, why do we see five layers in the neocortex? We propose that the different layers observed in the neocortex are all learning sequences using the same basic mechanism but the sequences learned in each layer are used in different ways. There is a lot we don’t understand about this, but we can describe the general idea. Before we do, it will be helpful to describe what the neurons in each layer connect to.

Diagram Page 60

The above diagram illustrates two neocortical regions and the major connections between them. These connections are seen throughout the neocortex where two regions project to each other. The box on the left represents a cortical region that is hierarchically lower than the region (box) on the right, so feed-forward information goes from left to right in the diagram. The down arrow projects to other areas of the brain. Feedback information goes from right to left. Each region is divided into layers. Layers 2 and 3 are shown together as layer 2/3.

The colored lines represent the output of neurons in the different layers. These are bundles of axons originating from the neurons in the layer. Recall that axons immediately split in two. One branch spreads horizontally within the region, primarily within the same layer. Thus all the cells in each layer are highly interconnected. The neurons and horizontal connections are not shown in the diagram.

There are two feed-forward pathways, a direct path shown in orange and an indirect path shown in green. Layer 4 is the primary feed-forward input layer and receives input from both feed-forward pathways.

**Layer 4 projects to layer 3.**

Layer 3 is also the origin of the direct feed-forward pathway. So the direct forward pathway is limited to layer 4 and layer 3.

Some feed-forward connections skip layer 4 and go directly to layer 3. And, as mentioned above, layer 4 disappears in regions far from sensory input. At that point, the direct forward pathway is just from layer 3 to layer 3 in the next region.

The second feed-forward pathway (shown in green) originates in layer 5. Layer 3 cells make a connection to layer 5 cells as they pass on their way to the next region. After exiting the cortical sheet, the axons from layer 5 cells split again. One branch projects to sub-cortical areas of the brain that are involved in motor generation. These axons are believed to be motor commands (shown as the down facing arrow). The other branch projects to a part of the brain called the thalamus which acts as a gate. The thalamus either passes the information onto the next region or blocks it.

Finally, the primary feedback pathway, shown in yellow, starts in layer 6 and projects to layer 1. Cells in layers 2, 3, and 5 connect to layer 1 via their apical dendrites (not shown). Layer 6 receives input from layer 5.

This description is a limited summary of what is known about layer to layer connections. But it is sufficient to understand our hypothesis about why there are multiple layers if all the layers are learning sequences.

Hypothesis on what the different layers do We propose that layers 3, 4 and 5 are all feed-forward layers and are all learning sequences. Layer 4 is learning first order sequences. Layer 3 is learning variable order sequences. And layer 5 is learning variable order sequences with timing. Let’s look at each of these in more detail.

**Layer 4**

It is easy to learn first order sequences using the HTM cortical learning algorithm. If we don’t force the cells in a column to inhibit each other, that is, the cells in a column don’t differentiate in the context of prior inputs, then first order learning will occur. In the neocortex this would likely be accomplished by removing an inhibitory effect between cells in the same column. In our computer models of the HTM cortical learning algorithm, we just assign one cell per column, which produces a similar result.

First order sequences are what are needed to form invariant representations for spatial transformations of an input. In vision, for example, x-y translation, scale, and rotation are all spatial transformations. When an HTM region with first order memory is trained on moving objects, it learns that different spatial patterns are equivalent. The resulting HTM cells will behave like what are called “complex cells” in the neocortex. The HTM cells will stay active (in the predictive state) over a range of spatial transformations.

At Numenta we have done vision experiments that verify this mechanism works as expected, and that some spatial invariance is achieved within each level. The details of these experiments are beyond the scope of this appendix.

Learning first order sequences in layer 4 is consistent with finding complex cells in layer 4, and for explaining why layer 4 disappears in higher regions of neocortex. As you ascend the hierarchy at some point it will no longer be possible to learn further spatial invariances as the representations will already be invariant to them.

**Layer 3**

Layer 3 is closest to the HTM cortical learning algorithm that we described in Chapter 2. It learns variable order sequences and forms predictions that are more stable than its input. Layer 3 always projects to the next region in the hierarchy and therefore leads to increased temporal stability within the hierarchy. Variable order sequence memory leads to neurons called “directionally-tuned complex cells” which are first observed in layer 3. Directionally-tuned complex cells differentiate by temporal context, such as a line moving left vs. a line moving right.

**Layer 5**

The final feed-forward layer is layer 5. We propose that layer 5 is similar to layer 3 with three differences. The first difference is that layer 5 adds a concept of timing. Layer 3 predicts “what” will happen next, but it doesn’t tell you “when” it will happen. However, many tasks require timing such as recognizing spoken words in which the relative timing between sounds is important. Motor behavior is another example; coordinated timing between muscle activations is essential. We propose that layer 5 neurons predict the next state only after the expected time. There are several biological details that support this hypothesis. One is that layer 5 is the motor output layer of the neocortex. Another is that layer 5 receives input from layer 1 that originates in a part of the thalamus (not shown in the diagram). We propose that this information is how time is encoded and distributed to many cells via a thalamic input to layer 1 (not shown in the diagram).

The second difference between layer 3 and layer 5 is that we want layer 3 to make predictions as a far into the future as possible, gaining temporal stability. The HTM cortical learning algorithm described in Chapter 2 does this. In contrast, we only want layer 5 to predict the next element (at a specific time). We have not modeled this difference but it would naturally occur if transitions were always stored with an associated time.

The third difference between layer 3 and layer 5 can be seen in the diagram. The output of layer 5 always projects to sub-cortical motor centers, and the feed-forward path is gated by the thalamus. The output of layer 5 is sometimes passed to the next region and sometimes it is blocked. We (and others) propose this gating is related to covert attention (covert attention is when you attend to an input without motor behavior). In summary, layer 5 combines specific timing, attention, and motor behavior. There are many mysteries relating to how these play together. The point we want to make is that a variation of the HTM cortical learning algorithm could easily incorporate specific timing and justify a separate layer in the cortex.

**Layer 2 and layer 6**

Layer 6 is the origin of axons that feed back to lower regions. Much less is known about layer 2. As mentioned above, the very existence of layer 2 as unique from layer 3 is sometimes debated. We won’t have further to say about this question now other than to point out that layers 2 and 6, like all the other layers, exhibit the pattern of massive horizontal connections and columnar response properties, so we propose that they, too, are running a variant of the HTM cortical learning algorithm.

**What does an HTM region correspond to in the neocortex?**

We have implemented the HTM cortical learning algorithm in two flavors, one with multiple cells per column for variable order memory, and one with a single cell per column for first order memory. We believe these two flavors correspond to layer 3 and layer 4 in the neocortex. We have not attempted to combine these two variants in a single HTM region. Although the HTM cortical learning algorithm (with multiple cells per column) is closest to layer 3 in the neocortex, we have flexibility in our models that the brain doesn’t have. Therefore we can create hybrid cellular layers that don’t correspond to specific neocortical layers. For example, in our model we know the order in which synapses are formed on dendrite segments. We can use this information to extract what is predicted to happen next from the more general prediction of all the things that will happen in the future. We can probably add specific timing in the same way. Therefore it should be possible to create a single layer HTM region that combines the functions of layer 3 and layer 5.

**Summary**

The HTM cortical learning algorithm embodies what we believe is a basic building block of neural organization in the neocortex. It shows how a layer of horizontally-connected neurons learns sequences of sparse distributed representations. Variations of the HTM cortical learning algorithm are used in different layers of the neocortex for related, but different purposes.

We propose that feed-forward input to a neocortical region, whether to layer 4 or layer 3, projects predominantly to proximal dendrites, which with the assistance of inhibitory cells, creates a sparse distributed representation of the input. We propose that cells in layers 2, 3, 4, 5, and 6 share this sparse distributed representation. This is accomplished by forcing all cells in a column that spans the layers to respond to the same feed-forward input.

We propose that layer 4 cells, when they are present, use the HTM cortical learning algorithm to learn first-order temporal transitions which make representations that are invariant to spatial transformations. Layer 3 cells use the HTM cortical learning algorithm to learn variable-order temporal transitions and form stable representations that are passed up the cortical hierarchy. Layer 5 cells learn variable-order transitions with timing. We don’t have specific proposals for layer 2 and layer 6. However, due to the typical horizontal connectivity in these layers it is likely they, too, are learning some form of sequence memory.

**Glossary**

Notes: Definitions here capture how terms are used in this document, and may have other meanings in general use. Capitalized terms refer to other defined terms in this glossary. Active State a state in which Cells are active due to Feed-Forward input Bottom-Up synonym to Feed-Forward Cells HTM equivalent of a Neuron Cells are organized into columns in HTM regions. Coincident Activity two or more Cells are active at the same time Column a group of one or more Cells that function as a unit in an HTM Region Cells within a column represent the same feed-forward input, but in different contexts. Dendrite Segment a unit of integration of Synapses associated with Cells and Columns HTMs have two different types of dendrite segments. One is associated with lateral connections to a cell. When the number of active synapses on the dendrite segment exceeds a threshold, the associated cell enters the predictive state. The other is associated with feed-forward connections to a column. The number of active synapses is summed to generate the feed-forward activation of a column. Desired Density desired percentage of Columns active due to Feed-Forward input to a Region The percentage only applies within a radius that varies based on the fan-out of feed-forward inputs. It is “desired” because the percentage varies some based on the particular input. Feed-Forward moving in a direction away from an input, or from a lower Level to a higher Level in a Hierarchy (sometimes called Bottom-Up) Feedback moving in a direction towards an input, or from a higher Level to a lower level in a Hierarchy (sometimes called Top-Down) First Order Prediction a prediction based only on the current input and not on the prior inputs – compare to Variable Order Prediction Hierarchical Temporal Memory (HTM) a technology that replicates some of the structural and algorithmic functions of the neocortex Hierarchy a network of connected elements where the connections between the elements are uniquely identified as Feed-Forward or Feedback HTM Cortical Learning Algorithms the suite of functions for Spatial Pooling, Temporal Pooling, and learning and forgetting that comprise an HTM Region, also referred to as HTM Learning Algorithms HTM Network a Hierarchy of HTM Regions HTM Region the main unit of memory and Prediction in an HTM An HTM region is comprised of a layer of highly interconnected cells arranged in columns. An HTM region today has a single layer of cells, whereas in the neocortex (and ultimately in HTM), a region will have multiple layers of cells. When referred to in the context of it’s position in a hierarchy, a region may be referred to as a level. Inference recognizing a spatial and temporal input pattern as similar to previously learned patterns Inhibition Radius defines the area around a Column that it actively inhibits Lateral Connections connections between Cells within the same Region Level an HTM Region in the context of the Hierarchy Neuron an information processing Cell in the brain In this document, we use the word neuron specifically when referring to biological cells, and “cell” when referring to the HTM unit of computation. Permanence a scalar value which indicates the connection state of a Potential Synapse A permanence value below a threshold indicates the synapse is not formed. A permanence value above the threshold indicates the synapse is valid. Learning in an HTM region is accomplished by modifying permanence values of potential synapses. Potential Synapse the subset of all Cells that could potentially form Synapses with a particular Dendrite Segment Only a subset of potential synapses will be valid synapses at any time based on their permanence value. Prediction activating Cells (into a predictive state) that will likely become active in the near future due to Feed-Forward input An HTM region often predicts many possible future inputs at the same time. Receptive Field the set of inputs to which a Column or Cell is connected If the input to an HTM region is organized as a 2D array of bits, then the receptive field can be expressed as a radius within the input space. Sensor a source of inputs for an HTM Network Sparse Distributed Representation representation comprised of many bits in which a small percentage are active and where no single bit is sufficient to convey meaning Spatial Pooling the process of forming a sparse distributed representation of an input One of the properties of spatial pooling is that overlapping input patterns map to the same sparse distributed representation. Sub-Sampling recognizing a large distributed pattern by matching only a small subset of the active bits in the large pattern Synapse connection between Cells formed while learning Temporal Pooling the process of forming a representation of a sequence of input patterns where the resulting representation is more stable than the input Top-Down synonym for Feedback Variable Order Prediction a prediction based on varying amounts of prior context – compare to First Order Prediction It is called “variable” because the memory to maintain prior context is allocated as needed. Thus a memory system that implements variable order prediction can use context going way back in time without requiring exponential amounts of memory.

**Efficiency of Predicted Sparseness as a Motivating Model for Hierarchical Temporal Memory**

**Part 1 - Introduction and Description.**

In any attempt to create a theoretical scientific framework, breakthroughs are often made when a single key “law” is found to underly what previously appeared to be a number of observed lesser laws. An example from Physics is the key principle of Relativity: that *the speed of light is a constant in all inertial frames of reference*, which quickly leads to all sorts of unintuitive phenomena like time dilation, length contraction, and so on. This discussion aims to do this for HTM by proposing that its key underlying principle is the *efficiency of predicted sparseness* at all levels. I’ll attempt to show how this single principle not only explains several key features of HTM identified so far, but also explains in detail how to model any required structural component of the neocortex.

The neocortex is a tremendously expensive organ in mammals, and particularly in humans, so it seems certain that the benefits it provides are proportionately valuable to the genes of an animal. We can use this relationship between cost and benefit, with sparseness and prediction as mediating metrics, to derive detailed design rules for the neocortex at every level, down to individual synapses and their protein machinery.

If you take one thing away from this talk, it should be that Sparse Distributed Representations are the key to Intelligence. <cite>Jeff Hawkins</cite>

**Sparse Distributed Representations** are a key concept in HTM theory. In any functional piece of cortex, only a small fraction of a large population of neurons will be active at a given time; each active neuron encodes some component of the semantics of the representation; and small changes in the exact SDR correspond with small differences in the detailed object or concept being represented. [Ahmad 2014](https://leanpub.com/realsmartmachines/read)describes many important properties of SDRs.

SDRs are one efficient solution to the problem of representing something with sufficient accuracy at optimal cost in resources, and in the face of ambiguity and noise. My thesis is that in forming SDRs, neocortex is striving to optimise a lossy compression process by representing only those elements of the input which are structural and ignoring everything else.

[Shannon](https://leanpub.com/realsmartmachines/read) proposed that any message has a concrete amount of information, measured in bits, which reflects the amount of surprise (i.e. something you couldn’t compute from the message so far, or by other means) contained in the message.

The most efficient message has zero length - it’s the message you don’t need to send. The next most efficient message contains only the information the receiver lacks to reconstruct everything the sender wishes her to know. Thus, by using memory and the right encoding to connect with it, a clever receiver (or memory system) can become very efficient indeed.

We will see that neocortex implements this idea literally, at all levels, as it attempts to represent, remember and predict events in the world as usefully as possible and at minimal cost.

The organising principle in cortical design is that components (from the whole organism down to a synapse) can do little about the amount of signal they receive, but they can - and do - adapt and learn to make best use of that signal to control what they do, only acting - sending a signal - when it’s the predicted optimal choice. This gives rise to sparseness in space and time everywhere, which directly reflects the degree of successful prediction present in any part of the system.

The success metric for a component in neocortex is the ratio of input data rate to output information rate, where the component has either a fixed minimum, or (for neurons and synapses) a fixed maximum, output level.

Deviations from the target indicate some failure to predict activity. This failure is either an opportunity to learn (and predict better next time), or, failing that, something which needs to be acted upon in some other way, by taking a different action or by passing new information up the hierarchy.

**Note** inputs in this context are any kind of signal coming in to the component under study. In the case of regions, layers and neurons, these include top-down feedback and lateral inputs as well as feedforward.

**Hierarchy**

Neocortex is a hierarchy because it has finite space to store its model of the world, and a hierarchy is an optimal strategy when the world itself has hierarchical structure. Each region in the hierarchy is subjected (by design) to a necessarily overwhelming rate of input, it will run at capacity to absorb its data stream, reallocating its finite resources to contain an optimal model of the world it perceives.

**Regions**

The memory inside a region of cortex is driven towards an “ideal” state in which it always predicts its inputs and thus produces a “perfect”, minimal message - containing its learned SDR of its world’s current state - as output. Any failure to predict is indicated by a larger output, the deviation from “ideal” representing the exact surprise of the region to its current perception of the world.

A region has several output layers, each of which has a different (and usually more than one) purpose.

For each region, two layers send (different) signals up the hierarchy, therefore signalling both the current state of its world and the encoding of its unpredictability. The higher region now gets details of something it should hopefully have the capacity to handle - predict - or else it passes the problem up the chain.

Two layers send (again different) signals down to lower layers and (in the case of motor) to subcortical systems. The content of these outputs will relate to the content as well as the stability and confidence of the region’s model, and also actions which are appropriate in terms of that content and confidence level.

**Layers**

A cortical layer which has fully predicted its inputs has a maximally sparse output pattern. A fully failing prediction pattern in a layer causes it to output a maximally bursting and minimally sparse pattern, at least for a short time. At any failure level in between, the exact evolution of firing in the bursting neurons encodes the precise pattern of prediction failure of the layer, and this is the information passed to other layers in the region, to other regions in cortex, or to targets outside the cortex.

The output of a cortical layer is thus a minimal message - it “starts” with the best match of its prediction and reality, followed (in a short period of time) by encodings of reality in the context of increasingly weak prediction.

**Columns**

A layer’s output, in turn, is formed from the combination of its neurons, which are themselves arranged in columns. The columnar arrangement of cells in cortical columns is the key design leading to all the behaviour described previously.

Pyramidal cells, which represent both the SDR activity pattern and the “memory” in a layer, are all contained in columns. The sparse pattern of activity across a layer is dictated by how all the cells compete within this columnar array.

Columns are composed of pyramidal cells, which act independently, and a complex of inhibitory cells which act together to define how the column operates. All cells share a very similar feedforward receptive field, due to the fact that feedforward axons physically run up through the narrow column and abut the pyramidal bodies as they squeeze past.

**Columnar Inhibition**

The inhibitory cells have a broader and faster feedforward response compared with the pyramidal cells [Reference](https://leanpub.com/realsmartmachines/read) so, in the absence of strong predictive inputs to any pyramidal cells, the entire assemblage of inhibitory neurons will be first to fire in a column. When this happens, these inhibitory cells excite those in adjacent columns, and a wave of inhibition spreads out from a successfully firing column.

The wave continues until it arrives at a column which has already been inhibited by a wave coming from elsewhere in the layer (from some recently active column). This gives rise to a pattern of inactivity around columns which are currently active.

**Predictive Activation**

Each cell in a column has its own set of feedforward and predictive inputs, so every cell has a different rate of depolarising as it is driven towards firing threshold.

Some cells may have received sufficient depolarising input from predictive lateral or top-down dendrites to reach firing threshold before the column’s sheath of inhibitory cells. In this case the pyramidal cell will fire first, trigger the column’s inhibitory sheath, and cause the wave of inhibition to spread out laterally in the layer.

**Vertical Inhibition in Columns**

When the inhibitory sheath fires, it also sends a wave of inhibitory signals vertically in the column. This wave will shut down any pyramidal cells which have not yet reached threshold, giving rise to a sparse activity pattern in the column.

The exact number of cells which get to fire before the sheath shuts them down depends mainly on how predictive each cell was and whether the sheath was triggered by a “winning cell” (previous section), by the sheath being first to fire, or as a result of neighbouring columns sending out signals.

If there is a wave of inhibition reaching a column, all cells are shut down and none (or no more) fire.

If there was a cell so predictive that it fired before the sheath, all other cells are very likely shut down and only one cell fires.

Finally, if the sheath was first to fire due to its feedforward input, the pyramidal cells are shut down quite quickly, but the most predictive may get the chance to fire just before being shut down.

This last process is called bursting, and gives rise to a short-lived pattern which encodes exactly how well the column as an ensemble has matched its predictions. Basically, the more cells which fire, the more “confused” the match between prediction and reality. This is because the inhibition happens quickly, so the gap between the first and last cell to burst must be small, reflecting similar levels of predictivity.

The bursting process may also be ended by an incoming wave of inhibition. The further away a competing column is, the longer that will take, allowing more cells to fire and extending the burst. Thus the amount of bursting also reflects the local area’s ability to respond to the inputs.

**Neurons**

Neurons are machines which use patterns of input signals to produce a temporal pattern of output signal. The neuron wastes most resources if its potential rises but just fails to fire, so the processes of adaption of the neuron are driven to a) maximise the response to inputs within a particular set, and b) minimise the response to inputs outside that set.

The set of excitatory inputs to one neuron are of two main types - feedforward and predictive; the number of each type of input varies from 10’s to 10’s of thousand; and the inputs arrive stochastically in combinations which contain mixtures of true structure and noise, so the “partitioning problem” a neuron faces is intractable. It simply learns to do the best it can.

Note that neurons are the biggest components in HTM which actually do anything! In fact, the regions, layers and columns are just organisational constructs, ways of looking at the sets of interacting neurons.

The neuron is the level in the system at which genetic control is exercised. The neuron’s shape, size, position in the neocortex, receptor selections, and many more things are decided per-neuron.

Importantly, many neurons have a genetically expressed “firing program” which broadly sets a target for the firing pattern, frequency and dependency setup.

Again, this gives the neuron an optimal pattern of output, and its job is to arrange its adaptations and learn to match that output.

**Dendrites**

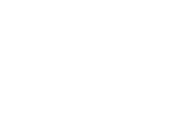
Distal dendrites have a similar but simpler and smaller scale problem of combining inputs and deciding whether to spike.

I don’t believe dendrites do much more than passively respond to global factors such as modulators and act as conduits for signals, both electrical and chemical, originating in synapses.

**Synapses**

Synapses are now understood to be highly active processing components, capable of growing both in size and efficiency in a few seconds, actively managing their response to multiple inputs - presynaptic, modulatory and intracellular, and self-optimising to best correlate a stream of incoming signals with the activity of the entire neuron.

*Part Two takes this idea further and details how a multilayer region uses the efficiency of predicted sparseness to learn a sensorimotor model and generate behaviour.*

[](https://leanpub.com/)

* **About**
* [About Leanpub](https://leanpub.com/about)
* [Blog](https://medium.com/@leanpub)
* [Team](https://leanpub.com/team)
* [Podcast](https://itunes.apple.com/ca/podcast/leanpub-podcast/id517117137)
* [Contact](https://leanpub.com/contact)
* [Press](https://leanpub.com/press)
* **Authors**
* [Why Leanpub](https://leanpub.com/authors)
* [Pricing](https://leanpub.com/pricing)
* [Manifesto](https://leanpub.com/manifesto)
* [Romance Authors](https://leanpub.com/romance)
* **Author Support**
* [Author Help](http://help.leanpub.com/author-help)
* [Getting Started](https://leanpub.com/help)
* [Manual](https://leanpub.com/help/manual)
* [API Docs](https://leanpub.com/help/api)
* **More**
* [Reader Help](http://help.leanpub.com/reader-help)
* [Publishers](https://leanpub.com/p)
* [Causes](https://leanpub.com/causes)
* **Legal**
* [Terms of Service](https://leanpub.com/terms)
* [Copyright Policy](https://leanpub.com/takedown)
* [Privacy Policy](https://leanpub.com/privacy)

Leanpub is copyright © 2010-2018 [Ruboss Technology Corp.](http://ruboss.com/) All rights reserved.