clortex

Clojure Library for Jeff Hawkins' Hierarchical Temporal Memory

---

Author: Fergal Byrne (fergalbyrnedublin@gmail.com)
Library: v0.1.1-SNAPSHOT
Date: 18 July 2014
Website: https://github.com/fergalbyrne/clortex
Generated By: MidjeDoc

---

1 Introduction
1.1 Background

2 Requirements
2.1 Directly Analogous to HTM/CLA Theory
2.2 Transparently Understandable Implementation in Source Code
2.3 Directly Observable Data
2.4 Sufficiently Performant
2.5 Useful Metrics
2.6 Appropriate Platform

3 System Architecture

4 Sensors
4.1 Numenta OPF Sensor (CSV data)
4.1.1 OPF Date Parsing

5 Encoders
5.1 Simple Scalar Encoder
5.2 Category/String Encoder
5.3 TODO

6 Data Structures and Functions

6.1 Neuron
6.2 Synapse
6.3 Axon
6.4 Dendrite
6.5 Column
6.6 Patch

7 License

---

1 Introduction

1.1 Background

Hierarchical Temporal Memory (*HTM*) is a theory of the neocortex developed by Jeff Hawkins in the early-mid 2000's. HTM explains the working of the neocortex as a hierarchy of **regions**, each of which performs a similar algorithm. The algorithm performed in each region is known in the theory as the **Cortical Learning Algorithm** (*CLA*).

Clortex is a reimagining and reimplementation of Numenta Platfrom for Intelligent Computing (NuPIC), which is also an Open Source project released by Grok Solutions (formerly Numenta), the company founded by Jeff to make his theories a practical and commercial reality. NuPIC is a mature, excellent and useful software platform, with a vibrant community, so please join us at Numenta.org.

Warning: pre-alpha software. This project is only beginning, and everything you see here will eventually be thrown away as we develop better ways to do things. The design and the APIs are subject to drastic change without a moment's notice.

Clortex is Open Source software, released under the GPL Version 3 (see the end of the README). You are free to use, copy, modify, and redistribute this software according to the terms of that license. For commercial use of the algorithms used in Clortex, please contact Grok Solutions, where they'll be happy to discuss commercial licensing.

2 Requirements

2.1 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 his 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, 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 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.

2.2 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.

2.3 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 theyâ€™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.

2.4 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 â€œproductionâ€ 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.

2.5 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.

2.6 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.

3 System Architecture

In order to mirror the HTM/CLA theory, `clortex` has a system architecture which is based on loosely-coupled components communicating over simple channels (or queues).

The primary dataflow in `clortex` involves 'enervation' data structures passing from *sources*, through a possible hierarchy of *regions* and flowing to a set of *sinks*.

Enervation, or inter-region communication, is encoded as a simple **SDR** map, which contains some very basic self-description data (`source`, `bit-size`, `description`, `type` etc) and a list of `on-bits`, `changed-on` and `changed-off` bit indices.

An SDR may also contain a channel which can be used to send the source (or an intermediary) data.

```
(def an-sdr {:source "a-uuid",
             :description "an example SDR",
             :type :scalar-encoding,
             :bit-size 512,
             :topology [512],
```

```
              :on-bits #{3, 22, 31, 55, 138},
              :changed-on #{22, 31, 138},
              :changed-off #{6, 111, 220},
              })
```

4 Sensors

Sensors gather information from the world and deliver it in encoded form to the CLA.

4.1 Numenta OPF Sensor (CSV data)

The first sensor in `clortex` reads CSV data which is compatible with Numenta's OPF (Online Prediction Framework) software.

```
gym,address,timestamp,consumption
string,string,datetime,float
S,,T,
Balgowlah Platinum,Shop 67 197-215 Condamine Street Balgowlah 2093,2010-07-02 00:00:00.0,5.3
Balgowlah Platinum,Shop 67 197-215 Condamine Street Balgowlah 2093,2010-07-02 00:15:00.0,5.5
Balgowlah Platinum,Shop 67 197-215 Condamine Street Balgowlah 2093,2010-07-02 00:30:00.0,5.1
Balgowlah Platinum,Shop 67 197-215 Condamine Street Balgowlah 2093,2010-07-02 00:45:00.0,5.3
```

The first line lists the field names for the data in the file. These field names are referenced elsewhere when specifying the field(s) which need to be predicted, or the encoders to use for that field. The second line describes the type fo each field (in Python terms). The third line is OPF-specific. `S` (referring to the `gym` field) indicates that this field, when it changes, indicates a new **sequence** of data records. The `T` (for the `timestamp` field) indicates that this field is to be treated as time-series data. These two concepts are important in powering the CLA's sequence learning.

4.1.1 OPF Date Parsing

```
(str (parse-opf-date "2010-07-02 08:15:00.01")) => "2010-07-02T08:15:00.010Z"
```

```
"after loading the hotgym data, it has 87840 items"
    (def hotgym-config {:file "resources/hotgym.csv"
```

```
                        :read-n-records :all
                        :fields ["gym" {:type :string
                                        :doc "Name of this Gym"
                                        :encoder {:type :hash-encoder
                                                  :bits 32
                                                  :on 8}
                                        :sequence-flag? true}
                                 "address" {:type :string
                                            :doc "Address of this Gym"
                                            :encoder {:type :hash-encoder
                                                      :bits 32
                                                      :on 8}}
                                 "timestamp" {:type :datetime
                                              :doc "Timestamp of this data record"
                                              :subencode [{:field :day-of-year}
                                                          {:field :day-of-week}
                                                          {:field :time-of-day}
                                                          {:field :weekday?}]}
                                 ]})
    (def hotgym (load-opf-file hotgym-config))
    (count (:parsed-data hotgym)) => 87840

    (mapv (comp str first) (nth (:parsed-data hotgym) 10)) =>
    ["Balgowlah Platinum" "Shop 67 197-215 Condamine Street Balgowlah 2093" "2010-07-02T02:30:00.000Z" "1.2"]

    (def encs (:encoders hotgym))


    (def enc-10 (data-encode hotgym 10))
    (def enc-11 (data-encode hotgym 11))

    (def enc-30 (data-encode hotgym 30))
enc-10 => #{8 12 13 14 15
            42 46 56 64 71 80 85 88
            96 99 101 102 104 118 123 124 130 136 141 145 163
            175 176 189 191 201 204 209 210 221 229 230 236
            237 239 242 246 287 288 289 290 291 292 293 294
```

```
           295 296 297 298 299 300 301 302 303 304 305 306
           307 429 430 431 432 433 434 435 436 437 438 439
           440 441 442 443 444 445 446 447 448 449 1348 1349
           1350 1357 1359 1360 1367 1375 1383 1393 1394 1395
           1401 1414 1425 1440 1443 1450 1453 1462 1466}

(difference enc-10 enc-11) => #{429 430 431 432 433 434 435 436 437 438}

(difference enc-10 enc-30) =>  #{429 430 431 432 433 434 435 436 437 438
                               439 440 441 442 443 444 445 446 447 448 449
                               1348 1349 1350 1359 1360 1375 1383 1393 1394
                               1395 1401 1414 1425 1440 1443 1462 1466}
```

```
#_(write-edn-file hotgym "resources/hotgym.edn")
```

5 Encoders

Encoders are very important in `clortex`. Encoders turn real-world data into a form which `clortex` can understand - **Sparse Distrubuted Representations** (or *SDRs*). Encoders for the human brain include retinal cells (or groups of them), as well as cells in the ear, skin, tongue and nose.

5.1 Simple Scalar Encoder

Encoders convert input data values into bit-array representations (compatible with Sparse Distributed Representations). The simplest encoder converts a scalar value into a bit-array as seen below. We'll set up a very small scalar encoder with 4 bits on out of 12, so we can see how it works (e.5.1).

e.5.1 - a very simple scalar encoder

```
(def enc (s/scalar-encoder :bits 12 :on 4)) ; uses default params min 0.0, max 100.0
```

`scalar-encoder` returns a map of functions which can be used in various parts of the encoding of data. We'll define those functions by pulling them out of the map (e.5.2):

e.5.2 - pulling out the functions

```
(def sencode (:encode enc))

(def sencoders (:encoders enc))

(def sencode-all (:encode-all enc))

(def encoder-record (s/->ScalarEncoder "field" 12 4 0.0 100.0 sencoders sencode))

(def sencode (.encode encoder-record))
```

Let's check that the bottom and top values give the bottom and top SDRs:

```
(sencode 0) => #{0 1 2 3}
(sencode 100) => #{8 9 10 11}
(sencode-all 0) =>
[[0 true]  [1 true]  [ 2 true]  [ 3 true]
 [4 false] [5 false] [ 6 false] [ 7 false]
 [8 false] [9 false] [10 false] [11 false]]
```

`scalar-encoder` defaults to NuPIC's scalar encoder parameters:

e.5.3 - default 21/127 bit encoder

```
(def enc (s/scalar-encoder)) ; uses default params 127 bits, 21 on, min 0.0, max 100.0

(def sencode (:encode enc))
```

```
(sencode 0) =>
#{ 0  1  2  3  4  5  6
```

```
   7  8  9 10 11 12 13
  14 15 16 17 18 19 20}
(count (sencode 0)) => 21
(sencode 50) =>
#{53 54 55 56 57 58 59
  60 61 62 63 64 65 66
  67 68 69 70 71 72 73}
(count (sencode 50)) => 21
(sencode 100) =>
#{106 107 108 109 110 111 112
  113 114 115 116 117 118 119
  120 121 122 123 124 125 126}
```

5.2 Category/String Encoder

We need to map arbitrary categories or string descriptors to bit encodings. Using the SHA1 message digest, we generate a list of 20 deterministically-generated bytes (this ensures the encoding will be consistent), which we use to fill the bit array until the required number of bits is set.

```
(def enc (h/hash-encoder :bits 12 :on 4))
(def cencode (:encode enc))
(def cencode-all (:encode-all enc))

(cencode "test") => #{0 2 3 8}
(cencode "another test") => #{2 4 6 11}
(cencode-all "test") =>
[[0 true] [1 false] [2 true]
 [3 true] [4 false] [5 false]
 [6 false] [7 false] [8 true]
 [9 false] [10 false] [11 false]]
```

5.3 TODO

- build a timestamp encoder
- build an adaptive scalar encoder

6 Data Structures and Functions

The design of `clortex` is based on large, homogenous, passive data structures (e.g. Layers) which are collections of simple structures (e.g. Neurons, Dendrites and Synapses), along with a set of simple functions which act on these data structures (e.g. `(cla/predictive? a-neuron)`).

6.1 Neuron

Neurons in `clortex` are represented as a map as follows:

```
(def a-neuron {:active 0,
               :activation-potential 0,
               :feedforward-potential 0,
               :predictive 0,
               :predictive-potential 0,
               :proximal-dendrite [#_[synapse ...]],
               :distal-dendrites [#_[dendrite...]],
               :axon nil
               })
```

Neurons with `:active` are designated as **active** neurons. Active neurons represent the layer's SDR, and also will send their signals to downstream neurons for prediction.

Simple functions act on neurons, such as `predictive?`, defined as follows:

```
(defn predictive?
  "checks if a neuron is in the predictive state"
  [neuron]
  (pos? (:predictive neuron)))

(defn set-predictive
  "sets a neuron's predictive state"
  [neuron p]
  (assoc-in neuron [:predictive] p))
```

and used like this:

neurons have simple predictive states

```
(predictive? a-neuron)

=> false
```

```
(predictive? (set-predictive a-neuron 1))

=> true
```

6.2 Synapse

Synapses represent connections between neurons.

6.3 Axon

Axons represent the terminal points of interneural connections. Patches (see below) maintain a set of axon endpoints to provide feedforward inputs from sensors or lower patches, as well as for in-patch predictive connections. Axons have a `source` and an `activity` value. Incoming SDR's are translated into values on the axon collection of a patch.

6.4 Dendrite

A dendrite is a set of synapses. Dendrites can either be *proximal* (meaning *near*) or *distal* (meaning *far*). Each neuron usually has one proximal dendrite, which gathers signals from feedforward sources, and many distal dendrites, which gather predictive signal, usually from nearby active neurons.

6.5 Column

A column is a vector of neurons, representing the vertical organisation of neurns in the neocortex. Columns share the same set of potential feedforward inputs, and also identify the spatial location of the neurons in a layer or region.

6.6 Patch

A Patch is a container for neurons and their connections (synapses). A patch is the system component which links to others and manages incoming and outgoing data, as well as the transformation of the memory in the synapses.

Patches are responsible for translating a pattern of activity on a set of inputs into the appropriate changes in the neurons.

7 License
