clortex 0.1.1-SNAPSHOT

** Clortex: Implementation in Clojure of Jeff Hawkins' Hierarchical Temporal Memory & Cortical Learning Algorithm. **Warning: Pre-alpha code.** This project has just begun as is under daily development. Anything and everything is likely to change drastically without a moment's notice.

dependencies

org.clojure/clojure	1.6.0
incanter/incanter-core	1.5.4
incanter/incanter-io	1.5.4
org.clojure/data.csv	0.1.2
org.clojure/data.json	0.2.1
enlive	1.1.5
clojure-opennlp	0.3.2
clojurewerkz/buffy	1.0.0-beta1
clj-time	0.6.0
com.stuartsierra/component	0.2.1
com.stuartsierra/flow	0.1.0
com.datomic/datomic-free	0.9.4578
expectations	2.0.6
quil	1.7.0
adi	0.1.5
lein-light-nrepl	0.0.17

namespaces

(this space intentionally left almost blank)

clortex.core

toc

Pre-alpha Entry points for Clortex as a library. Nothing to see here right now.

```
(ns clortex.core )
```

clortex.domain.encoders.core

toc

[Pre-alpha] Standard Encoders

The Cortical Learning Algorithm consumes data encoded as **Sparse Distributed Representations** (SDRs), which are arrays or matrices of binary digits (bits). The functions which convert values into SDRs are `clortex` **encoders**.

TODO: Factor out encoder functions. Use Graph or protocols?

```
(ns clortex.domain.encoders.core
  (:require [clortex.protocols :refer :all]
            [clortex.utils.hash :refer [sha1 mod-2]]))
```

`true` if `x` is within plus or minus `window` of `centre`

```
(defn within+-
    [x centre window]
    (and (> x (- centre window))
         (<= x (+ centre window))))
```

`true` if `bit` is in the first `on` bits

`true` if `bit` is in the last `on` bits (of `bits`)

```
(defn low-bit?  [bit on] (< bit on))
(defn high-bit?  [bit bits on] (<= bits (+ bit on)))
```

creates a bit encoder fn for the scalar encoder. the first `on` bits and the last `on` bits respond to inputs at the bottom and top of the encoder's range. other bits respond to values within a window of their centres.

```
(defn scalar-on?-fn
  [i min' max' bits on gap half-on w]
  (let [low-bit-off? (+ min' (* i gap))
        high-bit-off? (- max' (* (- bits i) gap))
        centre (+ min' (* (- i half-on) gap))]
    (if (low-bit? i on)
        #(<= % low-bit-off?)
        (if (high-bit? i bits on)
            #(> % high-bit-off?)
            #(within+- % centre (/ w 1.0))))))
```

```
(defrecord ScalarEncoder [field-name bits on-bits minimum maximum encoders encode]
  CLAEncoder)
```

constructs functions to encode scalars using a clamped linear sliding window

```
(defn scalar-encoder
  [& {:keys [minimum maximum bits on] :or {minimum 0.0 maximum 100.0 bits 127 on 21}}]
  (let [gap (/ (- maximum minimum) (- bits on))
        half-on (/ on 2)
        w (* gap half-on)
        encoders (mapv #(scalar-on?-fn % minimum maximum bits on gap half-on w) (range bits))
        encode-all (fn [x] (mapv #(vector % ((encoders %) x)) (range bits)))
        encode (fn [x] (set (mapv first (filter second (encode-all x)))))
        encode-to-bitstring (fn [x] (apply str (mapv #(if ((encoders %) x) 1 0) (range bits))))]
    {:encoders encoders
     :encode-all encode-all
     :encode encode
     :encode-to-bitstring encode-to-bitstring}))
```

makes a list of on-bits using the SHA1 hash of a string

```
(defn hash-bits
    [s len on]
    (loop [coll (sorted-set) bits (cycle (sha1 s)) bit 0]
      (let [step (first bits)                    ; skip step bits in the set
            bit (mod-2 (+ bit step) len)]        ; wrap around the set
        (if (= on (count coll))                  ; enough bits?
            coll
            (recur (conj coll bit) (next bits) bit)))))
```

converts non-nil/nil to true/false

```
(defn hash-on?-fn
    [i bits on]
    (fn [s] (if ((hash-bits s bits on) i) true false)))
```

constructs functions to encode values using a hash function

```
(defn hash-encoder
  [& {:keys [bits on] :or {bits 127 on 21}}]
  (let [truthy #(if % true false)
        encoders (vec (map #(hash-on?-fn % bits on) (range bits)))
        encode-all (fn [s] (let [hs (hash-bits s bits on)] (vec (map #(vec (list % (truthy (hs %))))
                                                                     (range bits)))))

        encode #(hash-bits % bits on)]
    {:encoders encoders
     :encode-all encode-all
     :encode encode}))
```

constructs functions to encode values using a hash function

```
(defn date-encoder
  [& {:keys [bits on] :or {bits 127 on 21}}]
  (let [truthy #(if % true false)
        encoders (vec (map #(hash-on?-fn % bits on) (range bits)))
        encode-all (fn [s] (let [hs (hash-bits s bits on)] (vec (map #(vec (list % (truthy (hs %))))
                                                                     (range bits)))))

        encode #(hash-bits % bits on)]
    {:encoders encoders
     :encode-all encode-all
     :encode encode}))
```

clortex.domain.encoders.rdse

toc

```
(ns clortex.domain.encoders.rdse
  (:use midje.sweet)
  (:require [clortex.utils.math :refer :all]
    [clojure.set :refer [difference union]]))
```

sorts buckets by their bottom value

```
(defn bottom-sorter
    [x y]
    (let [c (compare (x :bottom) (y :bottom))]
         (if (zero? c)
             (compare x y)
             c)))
```

```
(defn ordered-bins [bins] (sort-by :bottom bins))
```

returns the bucket which covers value. updates the 'read' slot of the bucket

```
(defn find-bucket
    [^double value buckets]
    (when-let [bucket (first (filter #(<= (:bottom %) value (:top %)) (:bins buckets)))]
        bucket))
```

returns a bucket map given centre, radius and index

```
(defn new-bucket
    [^double value ^double radius ^long index]
    {:bottom (- value radius) :top (+ value radius) :index index :counter 1 :read 0})
```

used to accumulate the best nearby bucket when searching

```
(defn min-distance
    [acc a-bucket]
    (let [diff (min (abs-diff (:mine acc) (:bottom a-bucket))
                    (:best acc))]
         (if (< diff (:best acc))
             (conj acc {:index (:index a-bucket) :best diff})
             acc)))
```

```
(defn sdrs [bins] (reduce conj #{} (map :sdr bins)))
(defn bottom-of-buckets [bins] (reduce min (map :bottom bins)))
(defn top-of-buckets [bins] (reduce max (map :top bins)))
(defn n-bins [buckets] (count (:bins buckets)))
```

```
(defn search-starter [bucket] {:index nil :best Integer/MAX_VALUE  :mine (:bottom bucket)})
(defn sdr->bitstring [sdr bits] (apply str (vec (map #(if (contains? (set sdr) %) 1 0) (range bits)))))
```

```
(defn new-sdr
    [bucket buckets]
    (let [bins (:bins buckets)
          ^int on (:on buckets)
          ^int bits (:bits buckets)
          randomer (:randomer buckets)]
         (if (empty? bins)
             (vec (range on))
             (let [sorted-bins (sort-by :bottom bins)
                   above? (> (:bottom bucket) (:bottom (first sorted-bins)))
                   nearest-buckets
                       (if above?
                           (vec (reverse (drop (- (count sorted-bins) on) sorted-bins)))
                           (vec (take on sorted-bins)))
                   nearest-bits (vec (sort (reduce #(union %1 (set (:sdr %2))) #{} nearest-buckets)))
                   previous-sdr (:sdr (first nearest-buckets))
                   previous-sdr (if above? previous-sdr (vec (reverse previous-sdr)))
                   remove-bit (previous-sdr (inc (randomer (dec on))))
                   remove-bit (previous-sdr (randomer on))
                   same-bits (vec (disj (set previous-sdr) remove-bit))
                   free-bits (vec (difference (set (range bits)) (set nearest-bits)))
                   new-bit-pos (randomer (count free-bits))
                   new-bit (free-bits new-bit-pos)
                   new-sdr (vec (sort (conj (set same-bits) new-bit)))]
                new-sdr))))
```

```
(defn add-to-buckets!
    [buckets bucket]
    (let [bits (:bits @buckets)
         sdr (new-sdr bucket @buckets)
         sdr-bucket (conj bucket {:sdr sdr})
         ;bitstring (sdr->bitstring sdr bits)
         ;sdr-bucket (conj bucket {:sdr sdr :bitstring bitstring})
         ]
       (swap! buckets update-in [:bins] conj sdr-bucket)
       sdr-bucket))
```

```
(defn add-bucket!
    [value buckets]
    (let [diameter (:diameter @buckets)
          radius (/ diameter 2.0)
          mn #(bottom-of-buckets (:bins @buckets))
          mx #(top-of-buckets (:bins @buckets))]
      (if (empty? (:bins @buckets))
        (let [bucket (new-bucket value radius (n-bins @buckets))]
          (add-to-buckets! buckets bucket))
        (do (while (> value (mx))
               (add-to-buckets! buckets (new-bucket (+ (mx) radius) radius (n-bins @buckets))))
            (while (< value (mn))
               (add-to-buckets! buckets (new-bucket (- (mn) radius) radius (n-bins @buckets))))))))
```

```
(defn random-sdr-encoder-1
    [& {:keys [^double diameter ^int bits ^int on] :or {diameter 1.0 bits 127 on 21}}]
    (let [randomer
            (random-fn-with-seed 123456)
          buckets
            (atom {:diameter diameter :bits bits :on on :randomer randomer :bins []})
          encode!
            (fn [^double x]
                (if-not (find-bucket x @buckets) (add-bucket! x buckets))
                (sort (:sdr (find-bucket x @buckets))))
          encode-to-bitstring!
            (fn [^double x]
                (sdr->bitstring (encode! x) bits))]
        {:buckets buckets
         :encode encode!
         :encode-to-bitstring! encode-to-bitstring!}))
```

clortex.domain.neuron toc

```
(ns clortex.domain.neuron
    (:require [clortex.utils.uuid :as uuid]
           [clortex.utils.math :refer :all]
           [clortex.domain.neuron.pure-neuron :as pure-n]
           [clortex.domain.patch.persistent-patch :as db-patch]
           [datomic.api :as d]))
```

```
(defn free-db []
  (let [uri "datomic:free://macbook.local:4334/patches"]
    (d/delete-database uri)
    (d/create-database uri)
    (let [conn (d/connect uri)
          schema (load-file "resources/datomic/schema.edn")]
      (d/transact conn schema)
      conn)))
```

creates a new dendrite

```
(defn dendrite
  []
  {})
```

adds an axon connection on neuron 'from' to neuron 'to'

```
(defn connect-axon
  [from to]
  (let [neuron to
         to-index (:neuron/index to)]
     (assoc-in neuron
               [:neuron/axon]
               {:axon/to to-index,
                :axon/signalled 0})))
```

retrieves the neuron at position pos in patch

```
(defn get-neuron
  [patch pos]
  (patch pos))
```

```
(comment
  (def p (neuron-patch 20))
  (count p)
  (def from (get-neuron p 3))
  (def to (get-neuron p 5))
  (connect-feedforward to from)
  (connect-axon from to)
(use 'clortex.neuron)
(def p (neuron-patch 1024))
(def c (/ (count p) 2))
(defn connections [patch n] (dotimes [i n] (let [x (get-neuron patch (rand-int c))
                        y (get-neuron patch (+ c (rand-int c))) ]
                    (connect-feedforward x y)
                    (connect-axon y x)))
      (println "done")))
```

clortex.domain.neuron.persistent-neuron toc

```
(ns clortex.domain.neuron.persistent-neuron
  (require [clortex.protocols :refer :all]))
```

```
(extend-type datomic.query.EntityMap PNeuron
  (neuron-index [this] (:neuron/index this))
  (neuron-id [this] (:neuron/uuid this))
  (distal-dendrites [this] (:neuron/distal-dendrites this))
  (proximal-dendrite [this] (:neuron/proximal-dendrite this))
  )
```

adds a feedforward synapse on neuron 'to' from neuron 'from'

```
(defn connect-feedforward
  [to from]
  (let [neuron to
         from-index (.neuron-index from)
         perm (/ (rand-int 256) 256.0)]
     (assoc-in neuron
               [:neuron/proximal-dendrite 0]
               {:synapse/pre-synaptic-neuron from-index,
                :synapse/permanence perm})))
```

clortex.domain.neuron.pure-neuron

toc

```
(ns clortex.domain.neuron.pure-neuron
  (:require [clortex.protocols :as cp]))
```

```
(defrecord PureNeuron
  [uuid index
   distal-dendrites proximal-dendrite
   active? predictive-potential activation-potential]
  cp/PNeuron
  (neuron-index [this] (:index this))
  (neuron-id [this] (:uuid this))
  (distal-dendrites [this] (:distal-dendrites this))
  (proximal-dendrite [this] (:proximal-dendrite this)))
```

```
(def empty-neuron
  (->PureNeuron nil -1 [] [] false 0 0))
```

returns a neuron (empty or merged with inps)

```
(defn neuron
  [& inps]
  (merge empty-neuron (if inps (apply hash-map inps) {})))
```

clortex.domain.patch.core toc

```
(ns clortex.domain.patch.core
    (:require [clortex.utils.math :refer :all]))
```

```
(defn make-columns
    [& {:keys [^int columns ^int cells-per-column dims] :or {columns 2048 cells-per-column 32 dims [2048]}}]
[])
```

```
(defn single-layer-patch
    [& {:keys [^int columns ^int cells-per-column dims] :as patch-spec :or {columns 2048 cells-per-column 32 dims [2048]}}]
    (let [randomer
            (random-fn-with-seed 123456)
          patch-columns (make-columns patch-spec)
          data
            (atom {:n-columns columns :cells-per-column cells-per-column :dims dims
                   :randomer randomer :patch patch-columns})]
        {:patch data}))
```

clortex.domain.patch.persistent-patch toc

```
(ns clortex.domain.patch.persistent-patch
  (:use [adi.utils :only [iid ?q]])
  (require [clortex.protocols :refer :all]
           [datomic.api :as d]
           [adi.core :as adi]))
```

```
(extend-type datomic.query.EntityMap PNeuronPatch
  (neurons [this] (:patch/neurons this))
  (neuron-with-index [this index]
   (filter #(= index (neuron-index %)) (neurons this)))
  (neuron-with-id [this id]
   (filter #(= id (neuron-id %)) (neurons this)))
  (columns [this] (:patch/columns this))
  (timestamp [this] (:patch/timestamp this))
  (set-input-sdr [this sdr] this)
  (connect-inputs [this] this)
  (feedforward-synapses [this] [])
  )
```

```
(defrecord DatomicPatch [patch-id patch conn]
  PNeuronPatch
  (neurons [this]
   (:patch/neurons patch))
  (neuron-with-index [this index]
   (filter #(= index (neuron-index %)) (neurons this)))
  (neuron-with-id [this id]
   (filter #(= id (neuron-id %)) (neurons this)))
  (columns [this] (:patch/columns patch))
  (timestamp [this] (:patch/timestamp patch))
  (set-input-sdr [this sdr] this)
  (connect-inputs [this] this)
  (feedforward-synapses [this] []))


(def empty-patch
  (->DatomicPatch nil nil nil))


(defn find-patch-id
  [ctx patch-uuid]
  (ffirst (d/q '[:find ?patch-id
                 :in $ ?p-uuid
                        :where [?patch-id :patch/uuid ?p-uuid]]
               (d/db (:conn ctx))
               patch-uuid)))


(defn load-patch [ctx ^DatomicPatch patch patch-id]
  (let [conn (:conn ctx)]
    (merge patch {:patch-id patch-id :conn conn :patch (d/entity (d/db conn) patch-id)})))


(defn load-patch-by-uuid [ctx patch-uuid]
  (let [conn (:conn ctx)
        patch-id (find-patch-id ctx patch-uuid)]
    (load-patch ctx empty-patch patch-id)))


(defn create-patch
  [ctx patch-uuid]
  (let [conn (:conn ctx)]
    @(d/transact conn [{:db/id (d/tempid :db.part/user)
                        :patch/uuid patch-uuid}])))


(defn create-adi-patch
  [ctx patch-uuid]
  (adi/insert! (:ds ctx) [{:patch {:uuid patch-uuid}}]))


      #_(let [uri "datomic:mem://adi-test"
            ds ds    (adi/datastore uri clortex-schema true true)
            _add  (adi/insert! ds [{:patch {:uuid patch-1}}])
            check (->> (adi/select ds {:patch/uuid patch-1})
                       first :patch :uuid)
            _tidy (d/delete-database uri)]
        check)
(defn find-patch-uuids
  [ctx]
  (let [conn (:conn ctx)]
    (d/q '[:find ?patch-uuid
           :where [_ :patch/uuid ?patch-uuid]]
         (d/db conn))))
```

```
(defn create-patch
  [ctx patch-uuid]
  (let [conn (:conn ctx)]
    @(d/transact conn [{:db/id (d/tempid :db.part/user)
                        :patch/uuid patch-uuid}])))


(defn find-neuron-id
  [ctx patch-id neuron-index]
  (ffirst (d/q '[:find ?neuron-id
                 :in $ ?patch ?neuron-index
                 :where [?patch :patch/neurons ?neuron-id]
                        [?neuron-id :neuron/index ?neuron-index]]
               (d/db (:conn ctx))
               patch-id
               neuron-index)))


(defn add-neuron
  [ctx patch-uuid]
  (let [conn (:conn ctx)
        patch-id (find-patch-id ctx patch-uuid)
        neurons (count
                 (d/q '[:find ?neuron
                        :in $ ?p-id
                        :where [?p-id :patch/neurons ?neuron]]
                      (d/db conn)
                      patch-id))
        neuron-id (d/tempid :db.part/user)]
    @(d/transact conn [{:db/id neuron-id
                        :neuron/index neurons
                        :neuron/feedforward-potential 0
                        :neuron/predictive-potential 0
                        :neuron/active? false}
                       {:db/id patch-id
                        :patch/neurons neuron-id}])))


(defn add-neurons-to
  [ctx patch-uuid n]
  (let [conn (:conn ctx)
        patch-id (find-patch-id ctx patch-uuid)
        neurons (count
                 (d/q '[:find ?neuron
                        :in $ ?p-id
                        :where [?p-id :patch/neurons ?neuron]]
                      (d/db conn)
                      patch-id))
        tx-tuples (for [i (range n)
                        :let [neuron-id (d/tempid :db.part/user)
                              neuron-index (+ i neurons)]]
                    [{:db/id neuron-id
                      :neuron/index neuron-index
                      :neuron/active? false}
                     {:db/id patch-id :patch/neurons neuron-id}])
        tx-data (reduce #(conj %1 (%2 0) (%2 1)) [] tx-tuples)]
    tx-data))


(defn add-inputs-to
  [ctx patch-uuid n]
  (let [conn (:conn ctx)
        ds (:ds ctx)
        patch-id (find-patch-id ctx patch-uuid)
        inputs (count
                 (d/q '[:find ?input
                        :in $ ?patch-id
                        :where
                        [?patch-id :patch/inputs ?dendrite]
                        [?dendrite :dendrite/synapses ?synapse]
                        [?synapse :synapse/pre-synaptic-neuron ?input]]
                      (d/db conn)
                      patch-id))
        dendrite-id (d/tempid :db.part/user)
        tx-dendrite (if (zero? inputs)
                      [{:db/id dendrite-id}
                       {:db/id patch-id :patch/inputs dendrite-id}]
                      [])
        tx-tuples (for [i (range n)
                        :let [input-id (d/tempid :db.part/user)
                              synapse-id (d/tempid :db.part/user)
                              input-index (+ i inputs)]]
                    [{:db/id input-id
                      :neuron/index input-index
                      :neuron/active? false}
                     {:db/id synapse-id
                      :synapse/pre-synaptic-neuron input-id
                      :synapse/permanence 1
                      :synapse/permanence-threshold 0}
                     {:db/id dendrite-id :dendrite/synapses synapse-id}])
        tx-data (reduce #(conj %1 (%2 0) (%2 1)) tx-dendrite tx-tuples)]
    (println tx-data)
    tx-data))


(defn add-neurons-to!
  [ctx patch-uuid n]
  @(d/transact (:conn ctx) (add-neurons-to ctx patch-uuid n)))


(defn add-inputs-to!
  [ctx patch-uuid n]
  @(d/transact (:conn (:ds ctx)) (add-inputs-to ctx patch-uuid n)))


(defn find-dendrites
  [ctx neuron-id]
  (let [conn (:conn ctx)]
    (d/q '[:find ?dendrite
           :in $ ?neuron
           :where [?neuron :neuron/distal-dendrites ?dendrite]]
         (d/db conn)
         neuron-id)))


(defn add-dendrite!
  [ctx neuron]
  (let [conn (:conn ctx)
        dendrite-id (d/tempid :db.part/user)]
    @(d/transact conn [{:db/id neuron :neuron/distal-dendrites dendrite-id}
                       {:db/id dendrite-id :dendrite/capacity 32}])
    ;(println "Added dendrite" dendrite-id "to neuron" neuron)
    (find-dendrites ctx neuron)))


(defn synapse-between
  [ctx patch-uuid from to]
  (let [conn (:conn ctx)
        patch-id (find-patch-id ctx patch-uuid)
        from-id (find-neuron-id ctx patch-id from)
        to-id (find-neuron-id ctx patch-id to)]
    ;(println "checking synapse from neuron " from-id "to" to-id)
    (d/q '[:find ?synapse
           :in $ ?to ?from
           :where
             [?to :neuron/distal-dendrites ?dendrite]
             [?dendrite :dendrite/synapses ?synapse]
             [?synapse :synapse/pre-synaptic-neuron ?from]]
         (d/db conn)
         to-id from-id)))



(defn connect-distal
  [ctx patch-uuid from to]
  (when (zero? (count (synapse-between ctx patch-uuid from to)))
   (let [conn (:conn ctx)
        randomer (:randomer ctx)
        patch-id (find-patch-id ctx patch-uuid)
        from-id (find-neuron-id ctx patch-id from)
        to-id (find-neuron-id ctx patch-id to)
        synapse-id (d/tempid :db.part/user)
        permanence-threshold 0.2
        permanent? (> (randomer 3) 0)
        permanence (* permanence-threshold (if permanent? 1.1 0.9))
        synapse-tx {:db/id synapse-id
                    :synapse/pre-synaptic-neuron from-id
                    :synapse/permanence permanence
                    :synapse/permanence-threshold permanence-threshold}
        dendrites (find-dendrites ctx to-id)
        dendrites (if (empty? dendrites)
                    (add-dendrite! ctx to-id)
                    dendrites)
        dendrite (ffirst dendrites)]
    ;(println "Connecting " from-id "->" to-id "Adding synapse" synapse-id "to dendrite" dendrite)
    @(d/transact conn [{:db/id dendrite :dendrite/synapses synapse-id}
                       synapse-tx]))))



(defn find-neurons
  [ctx patch-uuid]
  (let [conn (:conn ctx)
        patch-id (find-patch-id ctx patch-uuid)]
    (d/q '[:find ?neuron-index
           :in $ ?patch-id
           :where [?patch-id :patch/neurons ?neuron-id]
           [?neuron-id :neuron/index ?neuron-index]]
         (d/db conn)
         patch-id)))



(defn input-sdr
  [ctx patch-uuid]
  (let [conn (:conn ctx)]
    (d/q '[:find ?index ?active
           :in $ ?patch-uuid
           :where
           [?patch :patch/uuid ?patch-uuid]
           [?patch :patch/inputs ?dendrite]
           [?dendrite :dendrite/synapses ?synapse]
           [?synapse :synapse/pre-synaptic-neuron ?input]
           [?input :neuron/active? ?active]
           [?input :neuron/index ?index]]
         (d/db conn)
         patch-uuid)))
```

```
(defn find-neuron-ids
  [ctx patch-uuid]
  (let [conn (:conn ctx)
        patch-id (find-patch-id ctx patch-uuid)]
    (d/q '[:find ?neuron-id
           :in $ ?patch-id
           :where [?patch-id :patch/neurons ?neuron-id]]
         (d/db conn)
         patch-id)))
```

clortex.domain.patch.pure-patch toc

```
(ns clortex.domain.patch.pure-patch
  (:require [clortex.protocols :refer :all]
            [clortex.domain.neuron.pure-neuron :as n]))
```

```
(defrecord PurePatch [uuid neurons columns inputs outputs synapses timestamp]
  PNeuronPatch
  (neurons [this] (:neurons this))
  (neuron-with-index [this index]
   (filter #(= index (.neuron-index %)) (neurons this)))
  (neuron-with-id [this id]
   (filter #(= id (.neuron-id %)) (neurons this)))
  (set-neurons [this neurons] (assoc this :neurons neurons))
  (columns [this] (:columns this))
  (timestamp [this] (:timestamp this))
  (set-input-sdr [this sdr] this)
  (connect-inputs [this] this)
  (feedforward-synapses [this] []))
```

```
(comment
  (neurons [p] "returns a collection of the patch's neurons")
  (neuron-with-index [p index] "returns a neuron with given index (or nil)")
  (neuron-with-id [p id] "returns a neuron with given uuid (or nil)")
  (set-neurons [p neurons] "returns patch with neuron added")
  (columns [p] "collection of mini-columns")
  (timestamp [p])
  (set-input-sdr [p sdr] "returns a patch with inputs matched to sdr")
  (connect-inputs [p] "returns a patch with inputs connected to proximal dendrites")
  (feedforward-synapses [p] "returns a collection of neurons affected by inputs"))
```

```
(def empty-patch
  (->PurePatch nil [] [] [] [] {} -1))
```

returns a patch (empty or merged with inps)

```
(defn patch
  [& inps]
  (merge empty-patch (if inps (apply hash-map inps) {})))
```

clortex.domain.sensors.core

toc

[Pre-alpha] OPF-Style Sensors

Currently reads an OPF-style CSV file and converts it into Clojure data structures.

TODO: ?

```clojure
(ns clortex.domain.sensors.core
  #_(:refer-clojure :exclude [second extend])
  (:require [clojure.data.csv :as csv]
            [clojure.java.io :as io]
            [clojure.pprint :refer [pprint]]
            [clortex.domain.sensors.date :refer [parse-opf-date]]
            [clortex.domain.encoders.core :as enc]
            [clortex.domain.encoders.rdse :as rdse]))
```

converts a CSV item (a string) into a Clojure value

```clojure
(defn parse-opf-item
    [v t]
    (condp = t
      "datetime" (parse-opf-date v)
      "float" (double (read-string v))
      v))
```

converts a CSV item (a string) into a Clojure value. catches and throws exceptions

```clojure
(defn safe-parse-opf-item
    [v t]
    (try (parse-opf-item v t)
    (catch Exception e (do (println (str "caught exception for value " v)) (throw e)))))
```

```clojure
(defn parse-opf-row
    [line & {:keys [fields types flags]}]
    (vec (for [i (range (count line))]
      (let [^String v (line i) ^String t (types i) ^String field (fields i) ^String flag (flags i)
        parsed (parse-opf-item v t)
        opf-meta {:raw v :type t :field field :flag flag}]
        (with-meta
          [parsed]
          {:opf-meta opf-meta})))))
```

parse OPF data from CSV test rows

```clojure
(defn parse-opf-data
  [raw-csv & {:keys [fields types flags]}]
  (mapv #(parse-opf-row % :fields fields :types types :flags flags) (drop 3 raw-csv)))
```

converts a CSV item (a string) into a Clojure value

```clojure
(defn make-encoder
    [field encoder-type]
    (condp = encoder-type
      "datetime" (enc/date-encoder)
      "float" (rdse/random-sdr-encoder-1)
      (enc/hash-encoder)))
```

```clojure
(defn make-encoders
  [fields types]
  (loop [inputs [fields types] result []]
    (if (empty? (first inputs))
      result
      (recur [(rest (first inputs)) (rest (second inputs))]
             (conj result (make-encoder (ffirst inputs) (first (second inputs)))))))


(defn load-opf-data [data & n]
  (let [raw-csv (if n (vec (take (first n) data))
                  (vec data))
        fields (raw-csv 0)
        types (raw-csv 1)
        flags (raw-csv 2)
        encoders (make-encoders fields types)
        opf-map {:fields fields :types types :flags flags :encoders encoders}
        parsed-data (parse-opf-data raw-csv :fields fields :types types :flags flags)
        ]
    (println "loaded" (count raw-csv) "lines")
    {:raw-csv raw-csv :fields fields :types types :flags flags
     :parsed-data parsed-data
     :encoders encoders
     }))


(defn load-opf-file [config]
  (let [f (:file config)
        n (:read-n-records config)
        fileio  (with-open [in-file (io/reader f)]
                      (vec (doall (csv/read-csv in-file))))
        n (if (and n (not= n :all)) n (count fileio))]
    (println "loaded" (count fileio) "lines")
       (load-opf-data fileio n)))


(defn write-edn-file [data f]
  (with-open [out-file (io/writer f)]
    (.write out-file (pr-str data))))


(defn write-edn-file [data f]
  (with-open [out-file (io/writer f)]
    (pprint data out-file)))


(comment
    (def hotgym (load-opf-file "resources/hotgym.csv")))
```

clortex.domain.sensors.date toc

```
(ns clortex.domain.sensors.date
    (require [clj-time.core :as tc]
        [clj-time.format :as tf]))


(def opf-timestamp-re #"(\d{4})-(\d{2})-(\d{2}) (\d{2}):(\d{2}):([0-9.]+)")
(defn strip-leading-zeros [s] (clojure.string/replace-first s #"^0+([1-9.])" "$1"))
```

```
(defn old-parse-opf-date
    [s]
    (let [m (re-matches opf-timestamp-re s)]
      (if m (let [rev (reverse (map strip-leading-zeros (rest m)))
                  secs (java.lang.Double/parseDouble (first rev))
                  items (map #(. Integer parseInt %) (rest rev))
                  ]
        (apply tc/date-time (reverse (conj items secs)))))))
```

```
(def opf-format (tf/formatter "yyyy-MM-dd HH:mm:ss.SS"))
```

(tf/parse opf-format "16:13:49:06 on 2013-04-06")

```
(defn parse-opf-date [s] (tf/parse opf-format s))
```

clortex.encoders toc

```
(ns clortex.encoders)
```

toc

clortex.protocols toc

```
(ns clortex.protocols)
```

Protocol for basic patch (Layer/Region) operations

```
(defprotocol PNeuronPatch
  (neurons [p] "returns a collection of the patch's neurons")
  (neuron-with-index [p index] "returns a neuron with given index (or nil)")
  (neuron-with-id [p id] "returns a neuron with given uuid (or nil)")
  (set-neurons [p neurons] "returns patch with neuron added")
  (columns [p] "collection of mini-columns")
  (timestamp [p])
  (set-input-sdr [p sdr] "returns a patch with inputs matched to sdr")
  (connect-inputs [p] "returns a patch with inputs connected to proximal dendrites")
  (feedforward-synapses [p] "returns a collection of neurons affected by inputs"))
```

Protocol for Cortical Learning Algorithm Neurons

```
(defprotocol PNeuron
  (neuron-index [n] "index of this neuron within its patch")
  (neuron-id [n] "uuid of this neuron")
  (distal-dendrites [n] "collection of distal dendrites for this neuron")
  (proximal-dendrite [n]))
```

Protocol for CLA dendrites

```
(defprotocol PDendriteSegment
  (synapses [d])
  (add-synapse [d s])
  (capacity [d])
  (full? [d]))
```

```
(defprotocol PPersistable
  "Protocol for persistable object")
```

Encodes values into bit representations

```
(defprotocol CLAEncoder
  (bits [this] "returns width in bits of this encoder")
  (on-bits [this] "returns number of on-bits of this encoder")
  (field-name [this] "returns the field (using . to indicate hierarchy)")
  (encoders [this] "returns the bit encoder functions")
  (encode-all [this value] "returns a verbose data structure for an encoding of value")
  (encode [this value] "returns a set of on-bits encoding value"))
```

```
(extend-type Object
  CLAEncoder
  (bits [this] (:bits this))
  (on-bits [this] (:on-bits this))
  (field-name [this] (:field-name this))
  (encoders [this] (:encoders this))
  (encode-all [this value] (mapv #(vector % ((.encoders this %) value)) (range (.bits this))))
  (encode [this value] (set (vec (map first (filter second (.encode-all this value)))))))
```

clortex.core toc

```
(ns clortex.core)
```

clortex.utils.datomic toc

```
(ns clortex.utils.datomic
  (:require [datomic.api :as d]
            [adi.core :as adi]))
```

```
(def clortex-schema
  {:patch   {:type    [{:type :keyword}]
             :name    [{:type :string}]
             :uuid    [{:type :uuid}]
             :timestep [{:type :long :default 0}]
             :columns [{:type :ref
                        :ref  {:ns   :column
                               :rval :patch}
                        :cardinality :many}]
             :neurons [{:type :ref
                        :ref  {:ns   :neuron
                               :rval :patch}
                        :cardinality :many}]
             :inputs   [{:type :ref
                         :ref  {:ns   :dendrite
                                :rval :patch}
                         :cardinality :one}]}
   :column  {:type    [{:type :keyword}]
             :index   [{:type :long}]
             :neurons [{:type :ref
                        :ref  {:ns   :neuron
                               :rval :column}
                        :cardinality :many}]}
   :neuron  {:index                 [{:type :long}]
             :feedforward-potential [{:type :long
                                      :default 0}]
             :prediction-potential  [{:type :long
                                      :default 0}]
             :active?               [{:type :boolean
                                      :default false}]
             :proximal-dendrite     [{:type :ref
                                      :ref  {:ns :dendrite
                                             :rval :neuron}
                                      :cardinality :one}]
             :distal-dendrites      [{:type :ref
                                      :ref  {:ns :dendrite
                                             :rval :neuron}
                                      :cardinality :many}]}
   :dendrite {:type     [{:type :enum
                          :default :distal
                          :enum {:ns :dendrite.type
                                 :values #{:distal :proximal :input :output}}}]
              :capacity [{:type :long
                          :default 32}]
              :threshold [{:type :long
                           :default 16}]
              :active?  [{:type :boolean
                          :default false}]
              :synapses [{:type :ref
                          :ref   {:ns   :synapse
                                  :rval :dendrite}
                          :cardinality :many}]}
   :synapse {:type       [{:type :enum
                           :default :excitatory
                           :enum {:ns :synapse.type
                                  :values #{:excitatory :inhibitory :io}}}]
             :permanence [{:type :long
                           :default 0}]
             :permanence-threshold [{:type :long :default 0}]
             :source     [{:type :ref
                           :ref   {:ns   :neuron
                                   :rval :fanout}
                           :cardinality :one}]}
   })
```

clortex.utils.hash toc

```
(ns clortex.utils.hash
  (:import (java.security MessageDigest)))


(defn mod-2
  [num div]
  (let [m (rem num div)]
    (if (or (zero? m) (= (pos? num) (pos? div)))
      m
      (if (pos? div) (+ m div) m))))


(defn get-bytes [s] (byte-array (map (comp byte int) s)))


(defn sha1 [obj]
  (let [bytes (get-bytes (with-out-str (pr obj)))]
    (apply vector (.digest (MessageDigest/getInstance "SHA1") bytes))))
```

clortex.utils.math toc

```
(ns clortex.utils.math)


(defn factorial [n] (loop [i n val 1N] (if (= i 1) val (recur (dec i) (* i val)))))
#_(fact (factorial 3) => 6)
(defn binomial [n k] (/ (factorial n) (* (factorial (- n k)) (factorial k))))
#_(fact (binomial 3 2) => 3)
#_(fact (binomial 5 2) => 10)
(defn random-fn-with-seed [n]
    (let [r (java.util.Random. n)]
      (fn [m] (.nextInt r m))))


(defn abs-diff [x y] (. Math abs (- x y)))
```

clortex.utils.uuid toc

```
(ns clortex.utils.uuid)


(defn squuid []
  (let [uuid (java.util.UUID/randomUUID)
        time (System/currentTimeMillis)
        secs (quot time 1000)
        lsb (.getLeastSignificantBits uuid)
        msb (.getMostSignificantBits uuid)
        timed-msb (bit-or (bit-shift-left secs 32)
                          (bit-and 0x00000000ffffffff msb))]
    (java.util.UUID. timed-msb lsb)))
```

clortex.viz.core toc

```
(ns clortex.viz.core
  (:require [quil.core :refer :all]
            [datomic.api :as d]
            [clortex.domain.patch.persistent-patch :as patch]
            [clortex.utils.math :refer :all]))


(def uri "datomic:free://localhost:4334/patches")
(def conn (d/connect uri))


(defn neurons
  []
  (let [ctx {:conn conn}
        patch (ffirst (patch/find-patch-uuids ctx))
        neuron-data (patch/find-neuron-ids ctx patch)]
    (mapv #(d/entity (d/db conn) (first %)) neuron-data)))


(defn coords
  [i n-cells]
  (let [rows (int (Math/sqrt n-cells))
        scale (/ (height) rows 1.2)
        x (int (+ 20 (* scale (int (rem i rows)))))
        y (int (+ 20 (* scale (int (/ i rows)))))]
        [(* 2.0 x) y]))


(defn part-line
  [x y x1 y1 fraction]
  (line x y
        (+ x (* fraction (- x1 x)))
        (+ y (* fraction (- y1 y)))))


(defn draw-synapse
  [from-i n-cells synapse post-neuron]
  (let [permanence (:synapse/permanence synapse)
        permanence-threshold (:synapse/permanence-threshold synapse)
        i (:neuron/index post-neuron)
        connected? (>= permanence permanence-threshold)
        [x y] (coords i n-cells)
        [x2 y2] (coords from-i n-cells)]
    (do
      (stroke-weight 1.0)
      (stroke (if connected? 120 64))
      (part-line x2 y2 x y permanence)
      (part-line x y x2 y2 (/ permanence 10.0)))
    (if connected?
      (do
        (stroke 90 (* permanence 255) 255)
        (stroke-weight 1.0)
        (line x2 y2 x y)
        (when (:neuron/active? post-neuron)
          (stroke 90 200 125)
          (line x2 y2 x y))
        (stroke-weight 2.0)
        (stroke 120)
        (part-line x2 y2 x y permanence)))))



(defn draw-distals
  [i n-cells distals]
  (doall (for [distal distals
        synapse (:dendrite/synapses distal)]
    (let [from-neuron (:synapse/pre-synaptic-neuron synapse)
          to-neuron (:synapse/post-synaptic-neuron synapse)
          from-i (:neuron/index from-neuron)
          permanence (:synapse/permanence synapse)
          permanence-threshold (:synapse/permanence-threshold synapse)
          connected? (>= permanence permanence-threshold)
          active? (:neuron/active? from-neuron)
          predictive? (:neuron/active? to-neuron)
          [x y] (coords i n-cells)
          [x2 y2] (coords from-i n-cells)]
      ;(println "i" i "at (" x "," y ") to" from-i "at (" x2 "," y2 ")")
      (if active?
        (do
          (stroke-weight 1.0)
          (stroke (if connected? 120 64))
          (part-line x2 y2 x y permanence)
          (part-line x y x2 y2 (/ permanence 10.0))))
      (if predictive?
        (do
          (stroke 127 255 127)
          (stroke-weight 1.0)
          (stroke (if connected? 120 64))
          (part-line x2 y2 x y permanence)
          (part-line x y x2 y2 (/ permanence 10.0))))
      (if (and active? connected?)
        (do
          (stroke 90 (* permanence 255) (if connected? 255 64))
          (stroke-weight 1.0)
          (line x2 y2 x y)
          (stroke-weight 2.0)
          (stroke (if connected? 120 64))
          (part-line x2 y2 x y permanence)))))))



(defn draw-axons
  [i n-cells conn]
  (let [db (d/db conn)
        targets (d/q '[:find ?synapse ?post
                       :in $ ?i
                       :where
                       [?pre :neuron/index ?i]
                       [?post :neuron/distal-dendrites ?dendrite]
                       [?dendrite :dendrite/synapses ?synapse]
                       [?synapse :synapse/pre-synaptic-neuron ?pre]]
                     db
                     i)]
    ;(println "cell " i " enervates" (count targets) "cells")
    (doall (for [[synapse-id post-neuron-id] targets]
             #_(println "target of i" i "is" target)
             (let [synapse (d/entity db synapse-id)
                   post-neuron (d/entity db post-neuron-id)]
               (draw-synapse i n-cells synapse post-neuron))))))


(defn change-activations
  [cells]
  (let [current-pattern (mapv :neuron/active? cells)
        on-bits (count (filter true? current-pattern))
        target (/ (count cells) 40)
        active? #(>= target (random (count cells)))
        txs (vec (for [neuron cells]
             {:db/id (:db/id neuron) :neuron/active? (active?)}))]
    (d/transact conn txs)
    (neurons)
    ;(println current-pattern)))


(defn setup []
  (set-state! :randomer (random-fn-with-seed 123456))
  (smooth)                           ;; Turn on anti-aliasing
  (frame-rate 30)                    ;; Set framerate to 1 FPS
  (background 33))                   ;; Set the background colour to
```

Set the background colour to a nice shade of grey.

```
(defn choose-from-active
  [ctx cells]
  (let [randomer (:randomer ctx)
        n-cells (count cells)
        active-cells (vec (filter :neuron/active? cells))
        n-active (count active-cells)
        chosen-active (if (pos? n-active) (randomer n-active) (randomer n-cells))
        chosen-cells (if (pos? n-active) active-cells cells)
        chosen-neuron (get chosen-cells chosen-active)]
        (:neuron/index chosen-neuron)))


(defn draw []
  (let [randomer (state :randomer)
        ctx {:conn conn :randomer randomer}
        patch (ffirst (patch/find-patch-uuids ctx))
        cells (neurons)
        n-cells (count cells)
        rows (int (Math/sqrt n-cells))
        scale (/ (height) rows 1.2)
        diam (inc (int (* 0.2 scale)))
        changer (randomer 5)
        previous-sdr (vec (filter :neuron/active? cells))
        previously-active (choose-from-active ctx cells)
        cells (if (zero? changer) (change-activations cells) cells)
        new-sdr (vec (filter :neuron/active? cells))
        newly-active (if (zero? changer) (choose-from-active ctx cells) (randomer n-cells))
        active-cells (vec (filter :neuron/active? cells))
        n-active (count active-cells)
        ;_ (println "cells:" (count cells) "active:" active-cells)
        chosen-active (if (pos? n-active) (randomer n-active) (randomer n-cells))
        chosen-cells (if (pos? n-active) active-cells cells)
        chosen-neuron (get chosen-cells chosen-active)
        connect-from (:neuron/index chosen-neuron)
        connect-to (randomer n-cells)]
    (background (if (zero? changer) 0 44))
    (if (and
         (zero? (randomer 1))
         (not= newly-active previously-active))
      (do
        ;(println "connecting" connect-from "to" connect-to)
        (patch/connect-distal ctx patch previously-active newly-active)))
    #_(doseq [cell cells]
      (let [i (:neuron/index cell)
            distals (:neuron/distal-dendrites cell)
            fill-color (if (:neuron/active? cell) 255 66)
            [x y] (coords i n-cells)]
        ;(println "i" i "at (" x "," y ")")
        (stroke (randomer 64) (randomer 64) (randomer 64))             ;; Set the stroke colour to a random grey
        (stroke-weight 1)       ;; Set the stroke thickness randomly
        (fill (randomer (+ 100 (count distals))))                ;; Set the fill colour to a random grey
        (draw-distals i n-cells distals)
        ;(no-loop)))
    (doseq [cell previous-sdr]
      (let [i (:neuron/index cell)
            ]
        (draw-axons i n-cells conn)))
    (doseq [cell new-sdr]
      (let [i (:neuron/index cell)
            ]
        (draw-axons i n-cells conn)))
    (doseq [cell cells]
      (let [i (:neuron/index cell)
            fill-color (if (:neuron/active? cell) 255 66)
            [x y] (coords i n-cells)]
        ;(println "i" i "at (" x "," y ")")
        (stroke 127)             ;; Set the stroke colour to a random grey
        (stroke-weight 0.3)       ;; Set the stroke thickness randomly
        (fill fill-color 66 66)                ;; Set the fill colour to a random grey
        (rect x y diam diam)))
    (doseq [cell previous-sdr]
      (let [i (:neuron/index cell)
            fill-color 195
            [x y] (coords i n-cells)]
        ;(println "i" i "at (" x "," y ")")
        (stroke 127)             ;; Set the stroke colour to a random grey
        (stroke-weight 0.3)       ;; Set the stroke thickness randomly
        (fill fill-color 180 180)                ;; Set the fill colour to a random grey
        (rect x y diam diam)))
    #_(println (count cells) "\tcells"
             (count (filter :neuron/active? cells)) "\tactive")))



(defsketch example                  ;; Define a new sketch named example
  :title "Clortex Visualisation"    ;; Set the title of the sketch
  :setup setup                      ;; Specify the setup fn
  :draw draw                        ;; Specify the draw fn
  :size [800 400])                  ;; You struggle to beat the golden ratio
```

You struggle to beat the golden ratio