HTM / CLA v. 0.2

Rev. 1.1

Description Of Implementation

Contents

1 Classes of objects

Region The highest level entity which HTM operates in is the **Region**. The region takes a binary input vector from the previous region (or the "sensory" input directly) and outputs a binary vector to the region higher in the hierarchy (or simply outputs it). In addition to having getter methods (Python's *properties*) to access the data, `Region` class implements methods that manipulate the data in accordance with the HTM algorithm, i. e. to run HTM's spatial pooler one would call a `Region` instance's `spatial_pooler(·)` method.

The region can be thought of as a list of layers or as a list of columns each consisting of HTM **Cells**. If the layer is one-dimensional, we can represent the whole region as a

matrix:

$$\begin{bmatrix} \circ_{11} & \circ_{12} & \cdots & \circ_{1m} \\ \circ_{21} & \circ_{22} & & \circ_{2m} \\ \vdots & & \ddots & \vdots \\ \circ_{n1} & \circ_{n2} & \cdots & \circ_{nm} \end{bmatrix}$$

Then it is possible to switch between the two representations by simply transposing the matrix, i. e. the first HTM column would be a vector of $[\circ_{11}, \circ_{21}, \ldots, \circ_{n1}]$ elements. But the data structure used to store the region in the code is the array of *layers* of cells.

Cell Each cell keeps its layer's index and its positional index in that layer; it maintains a history of its own activation (one of inactive, active, predictive) and binary learning states.

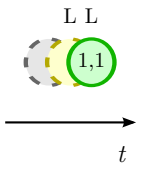

Figure 1: An active and learning HTM cell that was predictive and learning at t_{-1} and inactive at t_{-2}

Synapse Cells connect to each other and to the inputs by the means of Synapses. A synapse has a permanence value $p \in \mathbb{R}^{[0,1]}$; it is said to be *valid* if the synapse is above the threshold and *active* if it is valid and attached to an active cell / input.

Figure 2: A valid but inactive HTM synapse (due to the lack of input)

Synapses are located on two types of dendrites: proximal and distal.

Proximal dendrite This segment is connected to the inputs and is shared among all the cells in the column, so that all cells get the same input value. It is active if the amount of synapses is greater than a threshold.

Distal dendrite Every cell has a list of distal dendrites which are used to connect to other cells in the same layer. As proximal dendrites, a distal dendrite is considered active if the amount of active synapses is greater than the threshold. Additionally, it can be assigned *sequence* and *learning* flags which are used during the temporal pooling phase.

Both types of dendrites and the `Cell` class maintain some if its data in the form of time-history; this data structure is essentially `list` which only keeps limited amount of

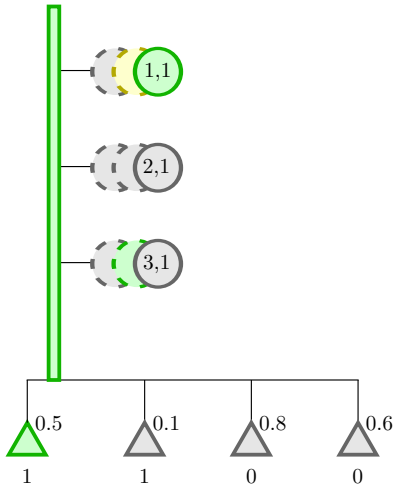

Figure 3: An active HTM proximal dendrite

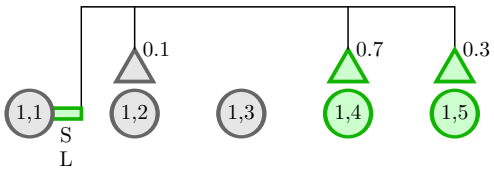

Figure 4: A cell with an active distal dendrite

states (on adding a new entry the oldest one is deleted) and indexes its elements with an offset of -1 to make some code look more natural, e. g. trying to obtain the current activation value (at time t_0) by `activation_hist[0]` would in fact return the element at index -1 in the list (the first one from the end) and so on.

2 Textual Representation

We don't discuss the types of data stored by each class of objects, instead we describe their textual representation that captures the most important properties of each class. This representation is vital to see what's actually going on with an HTM region:

```
<layer_idx>/<cell_idx>(<state><learning_flag>)~[<distal_dendrite>,
                                                <distal_dendrite>,...]
```

where:

`layer_idx` index of the neuron's layer (from the top)

`cell_idx` (sequential) index of the neuron's position in the layer

`state` one of the neuron's states:

`0` inactive

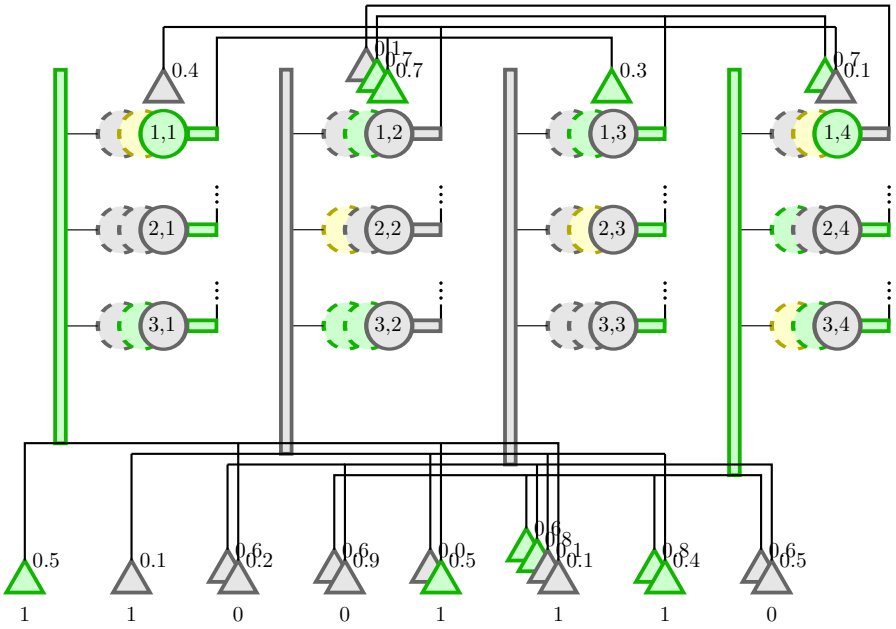

Figure 5: Full 3×4 HTM region

`1`	active
`2`	predictive

`learning_flag` is:

`L`	if the cell is learning
(empty)	otherwise

`distal_dendrite` a set of synapses on the neuron's distal dendrite segment:

```
<flags>[<synapse>, <synapse>, ...]
```

`flags`	is a potentially empty list of the following flags that correspond to certain segment's properties:
`*`	active segment
`L`	currently learning segment
`S`	sequence segment

`synapse` has the from:

```
<input_flag><idx>:<valid_flag><permanence>
```

`input_flag` is:

`=`	if the synapse gets the input

`?`	if it is unknown, whether it gets the input or not
(empty)	if the synapse doesn't get the input

`valid_flag` is:

`+`	if the synapse is valid (if the permanence value is above the threshold)
(empty)	otherwise

`permanence` real-valued permanence of the synapse in the interval $[0, 1]$

For example, consider the following neuron:

```
0/6(2)~[*[=0:+0.8, =2:+1.0, =3:+0.5, 4:+0.4, 9: 0.1, =13: 0.2],
         [ 10:+0.8, 7:+0.3, 12:+0.6, =14: 0.2, 15:+0.9]]
```

It's the seventh neuron in the first layer, it is currently in predictive state, it has two dendrite segments the first of which is active; first, second, third and sixth synapses are currently getting the input and all these synapses are valid (permanence value is more than 0.2) with the exception of the sixth neuron.

In addition to neurons with distal dendrites, there's a representation a proximal dendrite, which is very similar to distal dendrite*:*

```
<flags>[<synapse>, <synapse>, ...]
```

It has only one flag:

$*$	if the segment is active
(empty)	otherwise

3 Initialization

We start by creating the region of the specified size. Since we create proximal dendrites during initialization and they depend on the input size, it should be provided as well:

```
n_layers    = 3
n_cells     = 16
input_size  = 64
region = Region(n_layers, n_cells, input_size)
```

As we previously agreed, the region is the array of layers of cells (NumPy arrays are used to make the transposition easy):

```
self.region = np.array(
        [[Cell(c_idx, l_idx) for c_idx in range(n_cells)]
         for l_idx in range(n_layers)]
)
```

On region initialization, we assign a proximal dendrite to each column. The region should cover the whole input vector, so each proximal dendrite gets "input size / number

of cells per layer" *non-overlapping* inputs[1]. Those inputs are sampled uniformly at random without replacement from the input vector. The way HTM gets the input is similar to convolutional nets, where each neuron is connected to a local patch of the input; we randomize the input indices, however.

```
input_idxs = set(range(input_size))
n_syns = input_size / self.n_cells
for col_idx, column in enumerate(self.columns):
    potential_syn_idxs = random.sample(input_idxs, n_syns)
    input_idxs -= set(potential_syn_idxs)
    pd = ProximalDendrite(col_idx, potential_syn_idxs, self.n_cells,
                          init_boost, min_overlap)
    for cell in column:
        cell.set_prox_dendr(pd)
```

On proximal dendrite initialization, each of its synapses gets a permanence value that decreases linearly with the peak at the center of the input region, with the highest value being twice the connected permanence threshold and zero being the lowest one:

```
idxs = sorted(potential_syn_idxs, key=lambda n: abs(self.idx - n))
perms = np.linspace(CONNECTED_PERM_THR*2, 0, len(potential_syn_idxs))
self.potential_synapses = [Synapse(idx, permanence=perm)
                           for idx, perm in zip(idxs, perms)]
```

Continuing with the region initialization, we randomly interconnect cells within each layer with distal dendrite segments:

```
for layer in enumerate(self.layers):
    for cell in layer:
        for _ in range(n_dist_dendrites):
            cell_idxs = range(len(layer)); del cell_idxs[cell.idx]
            n_potential_syn = int(perc_interconnected * len(cell_idxs))
            potential_syn_idxs = set(
                    [random.choice(cell_idxs) for _ in range(n_potential_syn)]
            )
            potential_synapses = [Synapse(layer[idx]) for idx in potential_syn_idxs]
            cell.add_dist_dendr(DistalDendrite(potential_synapses))
```

This concludes the initialization of the region. Here's what we get for a 3×16 region:

```
[[ 0/0(0)~[[ 3:+0.4,  5:+0.4,  7:+0.5,  8:+0.8,  12:+0.4,  13:+0.3,  14:+0.3],
           [ 1: 0.2,  3:+0.8,  4:+1.0,  6:+0.6,  7:+0.3,  9:+0.7]],
   0/1(0)~[[ 0:+0.9,  4: 0.0,  5: 0.1,  9: 0.1,  12:+0.9,  14:+0.4,  15:+0.7],
           [ 8: 0.1,  9:+0.2,  10:+0.9,  11:+0.3,  12:+0.5,  14:+0.7]],
   0/2(0)~[[ 0:+0.8,  1:+0.8,  3:+0.2,  4:+0.9,  9:+1.0,  14:+0.9],
           [ 5: 0.2,  6:+0.6,  7:+0.8,  11:+0.4,  12:+0.6,  13:+0.9]],
   ...
   0/15(0)~[[ 1: 0.1,  2:+0.8,  5:+0.2,  6: 0.2,  10: 0.1,  11:+0.8,  14:+0.7],
            [ 0:+0.6,  2: 0.1,  4:+0.9,  7:+0.7,  13:+0.3,  14:+1.0]]],
 [ 1/0(0)~[[ 1:+0.2,  2:+0.7,  4:+0.3,  13:+0.4,  6: 0.1],
```

[1] This is done contrary to the Numenta's specifications.

```
          [ 1:+0.4,  3:+0.6,  13:+0.8,  14: 0.0,  7:+0.8]],
   1/1(0)~[[ 0:+0.9,  3:+0.5,  6: 0.0,  7:+0.4,  9: 0.2,  10:+0.4],
          [ 2: 0.1,  6:+0.4,  8: 0.1,  13: 0.1,  14:+0.7,  15:+0.6]],
   ...
 [ 2/0(0)~[[ 1: 0.2,  10:+0.6,  3:+0.8,  5:+0.7,  13: 0.1],
          [ 8:+0.6,  4:+0.3,  5:+0.3]],
   ...
   2/15(0)~[[ 0: 0.1,  4: 0.1,  6:+0.7,  8:+0.5,  12:+0.8,  14:+1.0],
            [ 1: 0.1,  6:+0.7,  10:+1.0,  12:+0.8,  13: 0.1,  14:+0.3]]]]
```

Since no cells are active, no synapses on distal dendrites are active, hence no distal dendrites are active and all the cells are in the inactive state. Note that since this the layer-oriented view on the region, we don't display proximal dendrites here.

4 Running

After the initialization, we follow the specifications as strictly as possible. On each iteration over the training examples (inputs), we connect the inputs to the region (give every synapse on each proximal dendrite a reference to the input vector) and do spatial and temporal poolers sequentially (for the testing purposes, we simply create a random binary vector as the input each trial):

```
perc_on     = 0.3
n_trials    = 6
output_hist = []
for t in range(n_trials):
    # For testing purposes: create random inputs (the correlation between inputs
    # at $t_{0}$ and $t_{1}$ should be zero)
    inputs = [int(random.uniform(0, 1) > (1 - perc_on)) for _ in range(input_size)]
    region.connect_inputs(inputs)
    region.spatial_pooler()
    region.temporal_pooler()
    # Keep track of the region's outputs
    output_hist.append(region.tp_output())
```

4.1 Spatial Pooler

Spatial pooler has three stages:

1. Activation of winners: during this stage we decide, which columns (proximal dendrites) will be active.

2. Learning: increase the permanence for those synapses on active proximal dendrites that get the input; decrease for those that don't.

3. Boosting: increase the boosting value and permanence of synapses on the proximal dendrites that don't fire often.

The output of the spatial pooler is a set active columns (proximal dendrites) that have learned synapses and that are boosted properly.

4.1.1 Activation of Winners

A synapse has a weight of 1 if its permanence is greater than the connection permanence threshold. If its weight is 1, it is said to *valid*. If it valid and if it either gets the input (the synapse on the proximal dendrite) or the cell with the distal dendrite on which the synapse is located is in the active state, then it is said to be *active*.

```
@property
def weight(self):
    if self.permanence >= CONNECTED_PERM_THR:
        return 1
    else:
        return 0

@property
def is_valid(self):
    return self.weight == 1

@property
def gets_input(self):
    return self.value > 0

@property
def value(self):
    if isinstance(self.obj, Cell):
        return self.cell.state
    else:
        return self.inputs[self.idx]

def is_active(self):
    if isinstance(self.obj, Cell):
        return self.is_valid & self.value == STATES['active']
    else:
        return self.is_valid & self.gets_input
```

The number of active synapses on each proximal dendrite is called *overlap*. However, if it's below the minimum overlap threshold, we scale that number by the region-specific boosting parameter:

```
@property
def _real_overlap(self):
    return len([s for s in self.active_synapses])

@property
def overlap(self):
    real_overlap = self._real_overlap
    if real_overlap < self.min_overlap:
```

```
        self.overlaps_hist.append(False)
        return 0
    else:
        self.overlaps_hist.append(True)
        return real_overlap * self.boost
```

Each proximal dendrite has a set of neighbours in a given radius:

```
def get_neighbour_idxs(self, radius):
    neigh = range(self.idx - radius, self.idx) + \
            range(self.idx + 1, self.idx + radius + 1)
    return [idx % self.n_cells for idx in neigh]
```

To decide which columns (proximal dendrites) should be active, for each proximal dendrite, we obtain overlap values of all its neighbours and activate the dendrite only if its own overlap value is greater than the overlap of the n'th neighbour, n given by the desired local activity global parameter:

```
def activate_winners(self, level=DESIRED_LOCAL_ACTIVITY):
    overlaps = self.overlaps
    for pd in self.prox_dendrites:
        neigh_idxs = pd.get_neighbour_idxs(self.inhib_radius)
        min_local_activity = sorted([overlaps[i] for i in neigh_idxs])[-level]
        if pd.overlap > min_local_activity:
            pd.set_active()
        else:
            pd.set_inactive()
```

Consider this scenario:

1. We get this input

   ```
   [1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0,
        0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0,
       1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0]
   ```

2. Some proximal dendrites have some synapses attached to active inputs, but their low permanence value makes them inactive and some valid synapses with high enough permanence but that don't get the input:

   ```
   [ 14:+0.4, 16:+0.3, =43: 0.1, 63: 0.0]
   ```

3. Some proximal dendrites are lucky enough to have synapses that are both valid and attached to active inputs. This particular dendrite has the overlap of two:

   ```
   *[=9:+0.4, =19:+0.3, =31: 0.1, 45: 0.0]
   ```

4. If there are no neighbours with higher overlap, the segment becomes active (and it did).

4.1.2 Learning

Learning is extremely easy once we get a set of active proximal dendrites. For each synapse of active proximal dendrite, if it gets the input we increase the permanence (so that the connection strengths) and decrease otherwise. If you think of, it works pretty much like Hebbian learning: Hebbian learning rule is $\Delta w_{ij} = \epsilon\, a_i a_j$, i. .e. the weights between the neurons are strengthened by ϵ is they both fire or both don't. In case of HTM, we may think of the proximal dendrite as always firing, and instead of having one learning rate ϵ, have two predefined parameters to increase and decrease the permanence (not the weight!) separately:

```
def sp_learn(self, perm_incr=PERM_INCR, perm_decr=PERM_DECR):
    for pd in self.active_pds:
        for syn in pd.potential_synapses:
            if syn.gets_input:
                syn.permanence += perm_incr
            else:
                syn.permanence -= perm_decr
```

While learning we also constrain the permanence to lie in $[0, 1]$:

```
@permanence.setter
def permanence(self, val):
    self._permanence = val
    self._permanence = min(1.0, max(0.0, self.permanence))
```

Consider the following case:

1. Before the learning we might have a proximal dendrite like this:

   ```
   *[=5:+0.4, =2:+0.3,  27: 0.1,  55: 0.0]
   ```

2. After the learning its permanence values should change accordingly:

   ```
   *[=5:+0.6, =2:+0.5,  27: 0.0,  55: 0.0]
   ```

4.1.3 Boosting

Each proximal dendrite maintains a history of its activations. We call the ratio of the number of activations to the total number of the time steps the *"firing rate"*.

```
def _get_rate(self, prop):
    return sum(prop) / float(len(prop))

@property
def firing_rate(self):
    return self._get_rate(self.activation_hist)
```

We call the value of 1% of the maximal firing rate among the neighbours of the segment the "*minimum duty cycle*". If the firing rate of a proximal dendrite is less than the minimum duty cycle, we increase the boosting value of the segment by the difference

between the minimum duty cycle and its own firing rate. Additionally, if the overlap rate of the segment is less than the minimum duty cycle, we increase the permanence values of all its synapses by the 10% of the connected permanence threshold:

```
def boost(self, connected_perm_thr=CONNECTED_PERM_THR):
    for pd in self.prox_dendrites:
        neighbours = [self.prox_dendrites[idx]
                      for idx in pd.get_neighbour_idxs(self.inhib_radius)]
        min_duty_cycle = 0.01 * max([n.firing_rate for n in neighbours])
        cur_firing_rate = pd.firing_rate
        if cur_firing_rate >= min_duty_cycle:
            pd.boost = 1
        else:
            pd.boost += (min_duty_cycle - cur_firing_rate)

        if pd.overlap_rate < min_duty_cycle:
            for syn in pd.potential_synapses:
                syn.permanence += 0.1 * connected_perm_thr
```

4.2 Temporal Pooler

Temporal pooler is more tricky. It consists of three consecutive steps:

1. Inferring active states: activate cells in active columns and choose the learning cell;
2. Inferring predictive states: put all laterally active cells into predictive state;
3. Learning: increase / decrease permanences of synapses on distal dendrites; form new synapses

Although learning exists as a separate step, we construct the synapse update list during the first two phases and then simply apply (or discard) it at the third phase.

4.2.1 Inferring Active States

Each distal dendrite has a list of potential synapses, some of which may be active. If the number of active synapses is greater than the activation threshold, we say that the dendrite is active (or was active at time step t):

```
def get_potential_synapses(self, t=0):
    return self.potential_synapses_hist[t]

def get_active_synapses(self, t=0, obey_perm_threshold=True):
    return [syn for syn in self.get_potential_synapses(t)
            if syn.is_active(obey_perm_threshold)]

def was_active(self, t=0, obey_act_threshold=True):
    n_valid_synapses = len(self.get_active_synapses(t))
    if obey_act_threshold:
        return n_valid_synapses > ACTIVATION_THR
```

```
    else:
        return n_valid_synapses > MIN_THR
```

Each cell should be able to select at least one active dendrite. If more than one dendrite is active, we return the sequence dendrite with the largest amount of active synapses (or just the one with the largest amount of synapses in no dendrites are sequence dendrites):

```
def get_active_dendrite(self, t=0):
    active_dds = [dd for dd in self.distal_dendrites if dd.was_active(t)]
    # FIXME: deal with zero active distal dendrites
    # XXX: the method is somewhat strange but follows Numenta's specifications
    if len(active_dds) == 0:
        raise Exception(msg)
    elif len(active_dds) == 1:
        return active_dds[0]
    else:
        active_dds.sort(key=lambda dd: dd.n_active_synapses(t))
        for dd in active_dds:
            if dd.was_sequence_dendrite(t):
                return dd
        return active_dds[-1]
```

Additionally, each distal dendrite may be a sequence dendrite; we maintain a history of these statuses while setting them by hand in the code. Same applies so learning states.

Activating cells In those columns that were activated by the spatial pooler, we look for *cells* to be activated. If a cell at one of the layers predicts the input (was in predictive state a quant of time ago) and one of its dendrites was an active and a sequence dendrite, the cell becomes active.

```
def infer_active_states(self):
    for pd in self.active_pds:
        is_predicted_input = False
        for cell in self.columns[pd.idx]:
            if cell.get_state(t=-1) == STATES['predictive']:
                try:
                    active_dendrite = cell.get_active_dendrite(t=-1)
                except:
                    break
                if active_dendrite.was_sequence_dendrite(t=-1):
                    is_predicted_input = True
                    cell.state = STATES['active']
```

In case none of the cells were in the predictive state, we say that the input was unanticipated and so later all the cells in the column become active.

```
        if not is_predicted_input:
            pd.activate_cells(self.columns)
```

Choosing the learning cell Additionally, if the active and sequential dendrite was also a learning dendrite one time step before, we set the *cell* into the learning state:

```
<...>
is_lc_chosen = False    # has the learning cell been chosen?
            <...>
            if active_dendrite.was_learning(t=-1):
                 is_lc_chosen = True
                 cell.is_learning = True
```

However, if there were no sequential and active dendrites, we need to choose the learning cell the other way. Choosing a pair of best matching cell and distal dendrite is a little complicated. For each cell, its best matching dendrite would be the one with the largest amount of synapses with attached inputs (e. g. active synapses but ignoring the permanence threshold):

```
def get_best_matching_dist_dendrite(self, t=0):
    active_dds = self.get_active_dist_dendrites(t, obey_act_threshold=False)
    if len(active_dds) == 0:
        return False
    else:
        active_dds.sort(
                key=lambda dd: dd.get_n_active_synapses(t, obey_perm_threshold=False)
        )
        return active_dds[-1]
```

This way we can select a pair of "cell"–"best matching dendrite" for each column. Among these pairs, we shall return one: either the one in which the dendrite has the largest amount of attached inputs, or (if there are no best matching dendrites) the one in which the cell has the smallest amount of dendrites

```
def get_best_matching_cell_dd(self, columns, t=0):
    column = columns[self.idx]
    cell_best_dd = [(cell, cell.get_best_matching_dist_dendrite(t))
                          for cell in column]
    if [pair[1] for pair in cell_best_dd].count(False) == len(cell_best_dd):
        # return a cell with the smallest amount of dds
        return (sorted(column, key=lambda cell: cell.n_distal_dendrites)[0],
                False)
    else:
        # dd with largest n of active synapses
        cell_best_dd.sort(
                key=lambda pair: pair[1] is not False and \
                                 pair[1].get_n_active_synapses(
                                         t, obey_perm_threshold=False
                                 )
        )
        return cell_best_dd[-1]
```

Having chosen the best matching cell, we set is to to the learning state:

```
<...>
```

```
        if not is_lc_chosen:
            (cell, dd) = pd.get_best_matching_cell_dd(self.columns, t=-1)
            cell.is_learning = True
```

Constructing update list If there was a distal dendrite in that pair, it should get a set of new synapses and become a sequence segment. The synapses selected for the update on the best matching dendrite are those active synapses which cells were in active state a time step ago. If the amount of those synapses it less than a "number of new synapses" parameter, we add compensate the difference by adding additional synapses: from those cells from the same layer as the best matching cell, that were learning cells a time step ago, we randomly select a cell, its distal dendrite and a potential synapse from that dendrite. That synapse also gets added to the update list:

```
            # sequence update
            if dd is not False:
                cand_new_cells = self._get_cand_new_cells(cell, t=-1)
                upd_synapses = dd.get_update_active_synapses(
                        t=-1, cand_new_cells=cand_new_cells
                        )
                dd.add_update(upd_synapses, is_sequence_segment=True)
```

where candidates for additional synapses are selected this way:

```
def _get_cand_new_cells(self, cell, t=0):
    return [c for c in self.layers[cell.layer_idx] if c.was_learning(t=-1)]
```

and the actual procedure for getting the update list on distal dendrite looks like this:

```
def get_update_active_synapses(self, t=0, cand_new_cells=False):
    active_synapses = [syn for syn in self.active_synapses
            if syn.cell.get_state(t) == STATES['active']]
    if cand_new_cells:
        n_new_syn = max(N_NEW_SYNAPSES - len(active_synapses), 0)
        for _ in range(n_new_syn):
            cell = random.choice(cand_new_cells)
            dd = random.choice(cell.distal_dendrites)
            syn = random.choice(dd.get_potential_synapses(t))
            active_synapses.append(syn)
    return active_synapses
```

4.2.2 Inferring Predictive States

A cell is laterally active if one of its distal dendrites is active. All laterally active cells in the region enter predictive state.

```
def infer_predictive_states(self):
    for cell in self.cells:
        if cell.is_laterally_active:
            cell.state = STATES['predictive']
```

Constructing update list Continuing iterating over all cells, for each active distal dendrite, we construct two update lists, then add them to the segment:

1. Predictive update: for an active distal dendrite with largest amount of active synapses one step ago of each cell, find update synapses similarly to how it was done while inferring the active states.

2. Active update: those active synapses on active distal dendrites of all cells, whose cell's states are currently active.

Finally, merge both updates as non-sequence segments:

```
for dd in cell.active_dist_dendrites:
    cand_new_cells = self._get_cand_new_cells(cell, t=-1)
    pred_dd = cell.get_best_matching_dist_dendrite(t=-1)
    pred_update = pred_dd.get_update_active_synapses(t=-1,
            cand_new_cells=cand_new_cells)
    active_update = dd.get_update_active_synapses(t=0, cand_new_cells=False)
    dd.add_update(active_update, is_sequence_segment=False)
    dd.add_update(pred_update, is_sequence_segment=False)
```

4.3 Learning

The last step of the temporal pooler is to apply the updates in two cases (assume that `upd_syns` contains a list of update synapses):

- if the cell is learning, then we increase the permanence values of update synapses on all its distal dendrites and decrease the permanence of all other potential synapses:

  ```
  for syn in upd_syns:
      syn.permanence += PERM_INCR
  for syn in set(dd.potential_synapses) - set(upd_syns):
      syn.permanence -= PERM_DECR
  ```

- if the cell was predicting a step ago, but not now (decrease the permanences).

  ```
  for syn in upd_syns:
      syn.permanence -= PERM_DECR
  ```

In addition, to each dendrite we add the difference between the update synapses and its existing potential synapses:

```
new_syn = set(upd_syns) - set(dd.potential_synapses)
for syn in new_syn:
    syn.permanence = INIT_PERM
    dd.potential_synapses.append(syn)
```

Spatial Pooler on a Set of Test Images

Since an HTM region only accepts binary vectors, there should be a set of encoders that translate various sorts of input data to binary vectors. Numenta doesn't publish

specification for the encoders. Nor does it specify how a region's reference behaviour. Hence it was difficult to compose a training set that would exploit temporal pooler's capability of doing temporal predictions. We limit ourselves to demonstrating the HTM's capabilities of forming the SDR.

For that task we've created two extremely simple datasets of five 16×16 pixels images each, one black-and-white, another one grayscale. The encoder we use is a trivial one: represent pixel's intensity in binary.

To get the SDR for an image, we run the algorithm for several trials over the data until the permanences reach their extreme values (0 or 1) or the trial limit has been reached. Since the region outputs a binary vector it may be converted into black-and-white image as well.

For example the region forms a stable representation of just one black-and-white image after five iterations:

```
INFO:root:Creating a 1 x 128 region...
INFO:root:Creating proximal dendrites with 16 synapses each...
INFO:root:Interconnecting cells with distal dendrites...
INFO:root: Level 0...
INFO:root:Region created.
INFO:root:1: |6| ?= 0
INFO:root:2: |9| != 1
INFO:root:3: |9| == 2
INFO:root:4: |9| == 3
INFO:root:5: |9| == 4
INFO:root:The region has learned the input, stopping.
```

What shows that the output of the region stabilized on the third iteration already; the total number of active / predictive neurons is nine a the final step.

Running the algorithm on the all five images gives different results:

```
...
INFO:root:=== Epoch 0 ===
INFO:root:1: |12| ?= 0
INFO:root:2: |16| != 1
INFO:root:3: |20| != 2
INFO:root:4: |24| != 3
INFO:root:5: |28| != 4
...
INFO:root:=== Epoch 98 ===
INFO:root:1: |33| ?= 0
INFO:root:2: |29| != 1
INFO:root:3: |15| != 2
INFO:root:4: |31| != 3
INFO:root:5: |26| != 4
INFO:root:=== Epoch 99 ===
INFO:root:1: |33| ?= 0
INFO:root:2: |29| != 1
INFO:root:3: |15| != 2
INFO:root:4: |31| != 3
```

```
INFO:root:5: |26| != 4
*[=150:+1.0, =236:+1.0, =254:+1.0, =306:+1.0,  607:+0.8, =884:+1.0,
    =940:+1.0, =1024:+1.0, =1141:+1.0, =1319:+1.0, =1494:+1.0,  1642: 0.0,
    1774: 0.0, =1779:+1.0, =1821:+1.0, =2047:+1.0]
*[=108:+1.0, =379:+1.0, =677:+1.0, =739:+1.0, =749:+1.0, =927:+1.0,
    =989:+1.0, =1071:+1.0,  1084:+0.2, =1158:+1.0, =1396:+1.0, =1659:+1.0,
    =1710:+1.0,  1766: 0.0, =1829:+1.0, =1960:+1.0]
...
```

The network almost stabilizes after 100 epochs but some of the weights have non-extreme values, so it keeps running. Other than that the SDR is quite stable.

Test runs on the grayscale set of images give similar result, except that the "convergence" happens faster (after 14 epochs already).