Chapter Revision History

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

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

Important Note to Readers:

The following algorithm concepts and details are adapted content from the original 2011 HTM whitepaper; the text and pseudocode have been edited to be consistent with the implementation of the Temporal Memory (TM) HTM Learning Algorithm in use at the time of BAMI release 0.4. Background information on both the SP and the TM algorithms is missing from release 0.4, and will be added over time.

Temporal Memory Algorithm Implementation and Pseudocode

As described in the "Spatial Pooler Algorithm Implementation and Pseudocode" section of this book, the Spatial Pooler forms a sparse distributed representation (SDR) of the input to a layer. This SDR represents columns of cells that received the most input (Fig. 1).

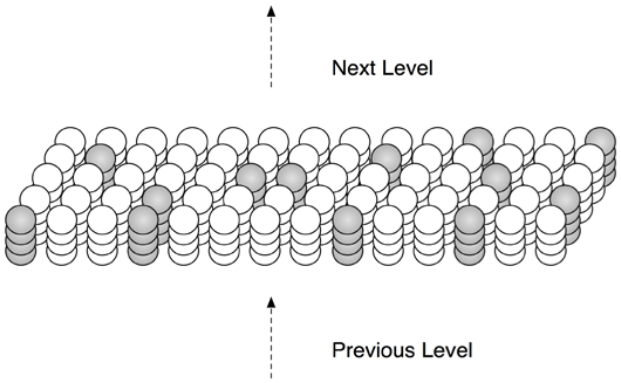

Figure 1 An HTM layer consists of columns of cells. Only a small portion of a layer is shown. Each column of cells receives activation from a unique subset of the input. Columns with the strongest activation inhibit columns with weaker activation. The result is a sparse distributed representation of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

The Temporal Memory algorithm does two things: it learns sequences of those SDRs formed by the Spatial Pooler and it makes predictions. Specifically it:

1) Forms a representation of the sparse input that captures the temporal context of previous inputs

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

We will discuss each of these steps in more detail.

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

After spatial pooling, a layer must convert the columnar representation of the input into a new representation that includes state, or context, from the past. The new representation is formed by activating a subset of the cells within each column, typically only one cell per column (Fig. 2).

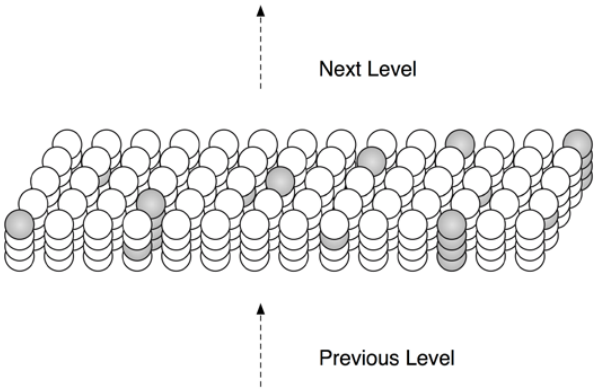

Figure 2 By activating a subset of cells in each column, an HTM layer can represent the same input in many different contexts. Columns only activate predicted cells. Columns with no predicted cells activate all the cells in the column. The figure shows some columns with one cell active and some columns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

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

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

Each column in an HTM layer consists of multiple cells. All cells in a column get the same feed-forward input. Each cell in a column can be active or not active. By selecting different active cells in each active column, we can represent the exact same input differently in different contexts. For example, say every column has 4 cells and the representation of every input consists of 100 active columns. If only one cell per column is active at a time, we have 4^100 ways of representing the exact same input. The same input will always result in the same 100 columns being active, but in different contexts different cells in those columns will be active. Now we can represent the same input in a very large number of contexts, but how unique will those different representations be? Nearly all randomly chosen pairs of the 4^100 possible patterns will overlap by about 25 cells. Thus two representations of a particular input in different contexts will have about 25 cells in common and 75 cells that are different, making them easily distinguishable.

The general rule used by an HTM layer is the following. When a column becomes active, it looks at all the cells in the column. If one or more cells in the column are already in the predictive state, only those cells become active. If no cells in the column are in the predictive state, then all the cells become active.

The system essentially confirms its expectation when the input pattern matches, by only activating the cells in the predictive state. If the input pattern is unexpected, then the system activates all cells in the column as if to say that all possible interpretations are valid.

If there is no prior state, and therefore no context and prediction, all the cells in a column will become active when the column becomes active. This scenario is similar to hearing the first note in a song. Without context you usually can't predict what will happen next; all options are available. If there is prior state but the input does not match what is expected, all the cells in the active column will become active. This determination is done on a column-by-column basis so a predictive match or mismatch is never an "all-or-nothing" event.

As mentioned in the terminology section above, HTM cells can be in one of three states. If a cell is active due to feed-forward input we just use the term "active". If the cell is active due to lateral connections to other nearby cells we say it is in the "predictive state" (Fig. 3).

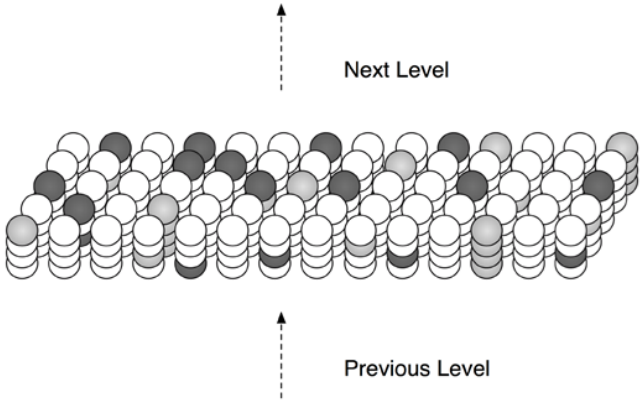

Figure 3 At any point in time, some cells in an HTM layer will be active due to feed-forward input (shown in light gray). Other cells that receive lateral input from active cells will be in a predictive state (shown in dark gray).

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

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

When a layer makes a prediction it depolarizes (i.e. puts into the predictive state) all the cells that will likely become active due to future feed-forward input. Because representations in a layer are sparse, multiple predictions can be made at the same time. For example if 2% of the columns are active due to an input, you could expect that ten different predictions could be made resulting in 20% of the columns having a predicted cell. Or, twenty different predictions could be made resulting in 40% of the columns having a predicted cell. If each column had ten cells, with one predicted at a time, then 4% of the cells would be in the predictive state.

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

How does a layer make a prediction? When input patterns change over time, different sets of columns and cells become active in sequence. When a cell becomes active, it forms connections to a subset of

the cells nearby that were active immediately prior. These connections can be formed quickly or slowly depending on the learning rate required by the application. Later, all a cell needs to do is to look at these connections for coincident activity. If the connections become active, the cell can expect that it might become active shortly and enters a predictive state. Thus the feed-forward activation of a set of cells will lead to the predictive state for other sets of cells that typically follow. Think of this as the moment when you recognize a song and start predicting the next notes.

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

We use the term “Temporal Memory” to describe the two steps of adding context to the representation and predicting. Now we will go into another level of detail. We start with concepts that are shared by the Spatial Pooler and Temporal Memory. Then we discuss concepts and details unique to the Spatial Pooler followed by concepts and details unique to the Temporal Memory.

Concepts Shared Between the Spatial Pooler and Temporal Memory Algorithms

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

Binary weights

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

Permanence

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

Dendrite segments

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

- A proximal dendrite segment forms synapses with feed-forward inputs. The active synapses on this type of segment are linearly summed to determine the feed- forward activation of a column.

- A distal dendrite segment forms synapses with cells within the layer. Every cell has many distal dendrite segments. If the sum of the active synapses on a distal segment exceeds a threshold, then the

associated cell enters the predicted state. Since there are multiple distal dendrite segments per cell, a cell's predictive state is the logical OR operation of several constituent threshold detectors.

Potential Synapses

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

Learning

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

Now we will discuss concepts specific to the temporal memory function.

Temporal Memory Algorithm Concepts

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

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

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

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

Every cell participates in many different distributed patterns and in many different sequences. A particular cell might be part of dozens or hundreds of temporal transitions. Therefore every cell has multiple dendrite segments, not just one. Ideally a cell would have one dendrite segment for each pattern of activity it wants to recognize. Practically though, a dendrite segment can learn connections for several completely different patterns and still work well. For example, one segment might learn 20 connections for each of 4 different patterns, for a total of 80 connections. We then set a threshold so the dendrite segment becomes active when any 15 of its connections are active. This introduces the possibility for error. It is possible, by chance, that the dendrite reaches its threshold of 15 active connections by mixing parts of different patterns. However, this kind of error is very unlikely, again due to the sparseness of the representations.

Now we can see how a cell with one or two dozen dendrite segments and a few thousand synapses can recognize hundreds of separate states of cell activity.

Temporal Memory Algorithm Details

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

1) For each active column, check for cells in the column that are in a predictive state, and activate them. If no cells are in a predictive state, activate all the cells in the column. The resulting set of active cells is the representation of the input in the context of prior input.

2) For every dendrite segment on every cell in the layer, count how many connected synapses correspond to currently active cells (computed in step 1). If the number exceeds a threshold, that dendrite segment is marked as active. Cells with active dendrite segments are put in the predictive state unless they are already active due to feed- forward input. Cells with no active dendrites and not active due to bottom-up input become or remain inactive. The collection of cells now in the predictive state is the prediction of the layer.

3) When a dendrite segment becomes active, modify the permanence values of all the synapses associated with the segment. For every potential synapse on the active dendrite segment, increase the permanence of those synapses that are connected to active cells and decrement the permanence of those synapses connected to inactive cells. These changes to synapse permanence are marked as temporary.

This modifies the synapses on segments that are already trained sufficiently to make the segment active, and thus lead to a prediction.

4) Whenever a cell switches from being inactive to active due to feed-forward input, we traverse each potential synapse associated with the cell and remove any temporary marks. Thus we update the permanence of synapses only if they correctly predicted the feed-forward activation of the cell.

5) When a cell switches from predictive state to inactive, undo any permanence changes marked as temporary for each potential synapse on this cell. This cell incorrectly predicted the feed-forward activation of the cell – in this case we slightly weaken the permanence values corresponding to previously active synapses.

Note that only cells that are active due to feed-forward input propagate their activity, otherwise predictions would lead to further predictions. As such the predictive state is a purely internal state of the cell.

First Order Versus Variable Order Sequences and Prediction

There is one more major topic to discuss before we end our discussion on temporal memory. What is the effect of having more or fewer cells per column? Specifically, what happens if we have only one cell per column?

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

If we increase to five cells per column, the available number of encodings of any particular input in our example would increase to 5^100, a huge increase over 4^100. In practice we have found using 10 to 16 cells per column to be sufficient for most situations.

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

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

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

Temporal and static recognition might appear to require different inference mechanisms. One requires recognizing sequences of patterns and making predictions based on variable length context. The other requires recognizing a static spatial pattern without using temporal context. An HTM layer with multiple cells per column is ideally suited for recognizing time-based sequences, and an HTM layer with one cell per column is ideally suited to recognizing spatial patterns.

Temporal Memory Pseudocode

Temporal memory pseudocode: inference alone

Phase 1

The first phase calculates the active state for each cell. For each winning column we determine which cells should become active. If the bottom-up input was predicted by any cell (i.e. its predictiveState was 1 due to a sequence segment in the previous time step), then those cells become active (lines 4-9). If the bottom-up input was unexpected (i.e. no cells had predictiveState output on), then each cell in the column becomes active (lines 10-12).

```
1.  for c in activeColumns(t)
2.
3.      buPredicted = false
4.      for i = 0 to cellsPerColumn - 1
5.      if predictiveState(c, i, t-1) == true then
6.              s = getActiveSegment(c, i, t-1, activeState)
7.              buPredicted = true
8.              activeState(c, i, t) = 1
9.
10.     if buPredicted == false then
11.             for i = 0 to cellsPerColumn - 1
12.                     activeState(c, i, t) = 1
```

Phase 2

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

```
13.  for c, i in cells
14.     for s in segments(c, i)
15.             if segmentActive(c, i, s, t) then
16.                     predictiveState(c, i, t) = 1
```

Temporal memory pseudocode: combined inference and learning

Phase 1

The first phase calculates the activeState for each cell that is in a winning column. For those columns, the code further selects one cell per column as the learning cell (learnState). The logic is as follows: if the bottom-up input was predicted by any cell (i.e. its predictiveState was 1), then those cells become active (lines 22-26). If that segment became active from cells chosen with learnState on, this cell is selected as the learning cell (lines 27-29). If the bottom-up input was not predicted, then all cells in the column become active (lines 31-33). In addition, the best matching cell is chosen as the learning cell (lines 35-39) and a new segment is added to that cell. Note that it is possible that there is no best matching cell; in this case getBestMatchingCell chooses a cell with the fewest number of segments, using a random tiebreaker.

```
17.  for c in activeColumns(t)
18.
19.     buPredicted = false
20.     lcChosen = false
21.     for i = 0 to cellsPerColumn - 1
22.             if predictiveState(c, i, t-1) == true then
23.                     s = getActiveSegment(c, i, t-1, activeState)
24.
25.                     buPredicted = true
```

```
26.                     activeState(c, i, t) = 1
27.                     if segmentActive(s, t-1, learnState) then
28.                         lcChosen = true
29.                         learnState(c, i, t) = 1
30.
31.     if buPredicted == false then
32.         for i = 0 to cellsPerColumn - 1
33.             activeState(c, i, t) = 1
34.
35.     if lcChosen == false then
36.         l,s = getBestMatchingCell(c, t-1)
37.         learnState(c, i, t) = 1
38.         sUpdate = getSegmentActiveSynapses (c, i, s, t-1, true)
39.         segmentUpdateList.add(sUpdate)
```

Phase 2

The second phase calculates the predictive state for each cell. A cell will turn on its predictive state output if one of its segments becomes active, i.e. if enough of its lateral inputs are currently active due to feed-forward input. In this case, the cell queues up the following changes: reinforcement of the currently active segment (lines 45-46).

```
40. for c, i in cells
41.     for s in segments(c, i)
42.         if segmentActive(s, t, activeState) then
43.             predictiveState(c, i, t) = 1
44.
45.             activeUpdate = getSegmentActiveSynapses (c, i, s, t, false)
46.             segmentUpdateList.add(activeUpdate)
```

Phase 3

The third and last phase actually carries out learning. In this phase segment updates that have been queued up are actually implemented once we get feed- forward input and the cell is chosen as a learning cell (lines 48-50). Otherwise, if the cell incorrectly predicted its activity, we negatively reinforce the segments (lines 51-53).

```
47.  for c, i in cells
48.     if learnState(s, i, t) == 1 then
49.         adaptSegments (segmentUpdateList(c, i), true)
50.         segmentUpdateList(c, i).delete()
51.     else if activeState(c, i, t) == 0 and predictiveState(c, i, t-1)==1 then
52.         adaptSegments (segmentUpdateList(c,i), false)
53.         segmentUpdateList(c, i).delete()
```

Temporal Memory Terminology, Data Structures and Routines

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

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

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

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

cell(c,i)	A list of all cells, indexed by i and c.
cellsPerColumn	Number of cells in each column.
activeColumns(t)	List of column indices that are winners due to bottom-up input (this is the output of the spatial pooler).
activeState(c, i, t)	A boolean vector with one number per cell. It represents the active state of the column c cell i at time t given the current feed-forward input and the past temporal context. activeState(c, i, t) is the contribution from column c cell i at time t. If 1, the cell has current feed-forward input as well as an appropriate temporal context.
predictiveState(c, i, t)	A boolean vector with one number per cell. It represents the prediction of the column c cell i at time t, given the bottom-up activity of other columns and the past temporal context. predictiveState(c, i, t) is the contribution of column c cell i at time t. If 1, the cell is predicting feed-forward input in the current temporal context.
learnState(c, i, t)	A boolean indicating whether cell i in column c is chosen as the cell to learn on.
activationThreshold	Activation threshold for a segment. If the number of active connected synapses in a segment is greater than activationThreshold, the segment is

	said to be active.
learningRadius	The area around a temporal memory cell from which it can get lateral connections.
initialPerm	Initial permanence value for a synapse.
connectedPerm	If the permanence value for a synapse is greater than this value, it is said to be connected.
minThreshold	Minimum segment activity for learning.
newSynapseCount	The maximum number of synapses added to a segment during learning.
permanenceInc	Amount permanence values of synapses are incremented when activity-based learning occurs.
permanenceDec	Amount permanence values of synapses are decremented when activity-based learning occurs.
predictedSegmentDec	If a cell incorrectly predicted its activity, the permanence values of previously active synapses on previously active segments are decremented by this amount.
segmentUpdate	Data structure holding three pieces of information required to update a given segment: a) segment index (-1 if it's a new segment), b) a list of existing active synapses, and c) a flag indicating whether this segment should be marked as a sequence segment (defaults to false).
segmentUpdateList	A list of segmentUpdate structures. segmentUpdateList(c,i) is the list of changes for cell i in column c.

The following supporting routines are used in the above code:

segmentActive(s, t, state)

This routine returns true if the number of connected synapses on segment s that are active due to the given state at time t is greater than activationThreshold. The parameter state can be activeState, or learnState.

getActiveSegment(c, i, t, state)

For the given column c cell i, return a segment index such that segmentActive(s,t, state) is true. If multiple segments are active, sequence segments are given preference. Otherwise, segments with most activity are given preference.

getBestMatchingSegment(c, i, t)

For the given column c cell i at time t, find the segment with the largest number of active synapses. This routine is aggressive in finding the best match. The permanence value of synapses is allowed to be below connectedPerm. The number of active synapses is allowed to be below activationThreshold, but must be above minThreshold. The routine returns the segment index. If no segments are found, then an index of -1 is returned.

getBestMatchingCell(c)

For the given column, return the cell with the best matching segment (as defined above). If no cell has a matching segment, then return a cell with the fewest number of segments using a random tiebreaker.

getSegmentActiveSynapses(c, i, t, s, newSynapses= false)

Return a segmentUpdate data structure containing a list of proposed changes to segment s. Let activeSynapses be the list of active synapses where the originating cells have their activeState output = 1 at time step t. (This list is empty if s = -1 since the segment doesn't exist.) newSynapses is an optional argument that defaults to false. If newSynapses is true, then newSynapseCount - count(activeSynapses) synapses are added to activeSynapses. These synapses are randomly chosen from the set of cells that have learnState output = 1 at time step t.

adaptSegments(segmentList, positiveReinforcement)

This function iterates through a list of segmentUpdate's and reinforces each segment. For each segmentUpdate element, the following changes are performed. If positiveReinforcement is true then synapses on the active list get their permanence counts incremented by permanenceInc. All other synapses get their permanence counts decremented by permanenceDec. If positiveReinforcement is false, then synapses on the active list get their permanence counts decremented by predictedSegmentDec. After this step, any synapses in segmentUpdate that do yet exist get added with a permanence count of initialPerm.