## 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 |
| 0.5 | Feb 2017 | Update to current algorithms | M. Lewis |

## Important Note to Readers:

The following text gives details of the Temporal Memory algorithm, including pseudocode and parameters. We highly recommend that you read about the Spatial Pooling algorithm first, and access some of the other Temporal Memory resources available in order to understand the high-level concepts and role of Temporal Memory in biology, and in HTM. You can find links to the latest Temporal Memory resources in the Temporal Memory chapter.

## Temporal Memory Algorithm

The Temporal Memory algorithm does two things: it learns sequences of Sparse Distributed Representations (SDRs) formed by the Spatial Pooling algorithm, and it makes predictions. Specifically, the Temporal Memory algorithm:

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, but first we present some important terms and concepts.

### Terminology

- Column: An HTM region is organized in columns of cells.
- Mini-column: See “Column”.
- Permanence value: A scalar value (0.0 to 1.0) that is assigned to each synapse to indicate how permanent the connection is. When a connection is reinforced, its permanence value is increased. Under other conditions, the permanence value is decreased.
- Permanence threshold: If a synapse’s permanence value is above this threshold, it is considered fully connected. Acceptable values are [0,1].
- Synapse: A junction between cells. A synapse can be in the following states:
    - Connected—permanence is above the threshold.
    - Potential—permanence is below the threshold.
    - Unconnected—does not have the ability to connect.

## Concepts Shared Between the Spatial Pooling and Temporal Memory Algorithms

### 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.

### 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

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 the permanence threshold.

### Learning

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

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 Pooling and Temporal Memory algorithms. The details for the Temporal Memory algorithm are described below.

Now we will discuss concepts specific to the Temporal Memory algorithm.

## Temporal Memory Algorithm Concepts

The Temporal Memory algorithm learns sequences and makes predictions. In the Temporal Memory algorithm, 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.

### Use of Sparse Distributed Representation Properties

It is worth noting a few important properties of Sparse Distributed Representations (SDRs) that the Temporal Memory algorithm 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.

### First Order Versus Variable Order Sequences and Prediction

What is the effect of having more or fewer cells per column? Specifically, what happens if we have only one cell per column?

As we show later in this chapter, 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 Algorithm Steps

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

The input to the Temporal Memory algorithm is sequences of Sparse Distributed Representations (SDRs) that are formed by the Spatial Pooling algorithm. As described in the "Spatial Pooling Algorithm" chapter of this book, the Spatial Pooling algorithm 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).

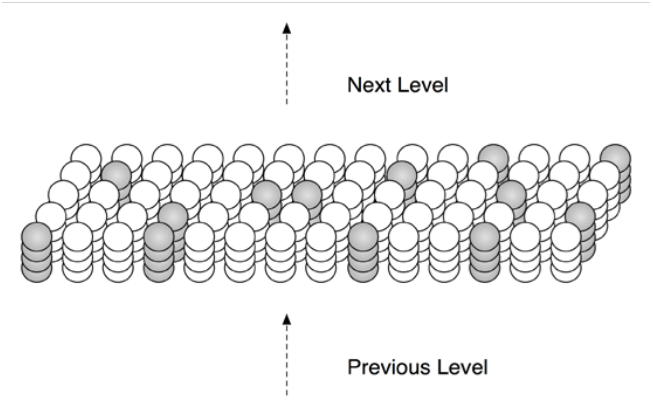


*Figure 1* Depiction of the Spatial Pooling algorithm. An HTM layer consists of columns of cells. Only a small portion of a layer is shown. Each column of cells receives activation from a unique subset of the input. Columns with the strongest activation inhibit columns with weaker activation. The result 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.)

After Spatial Pooling, the Temporal Memory algorithm converts 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).

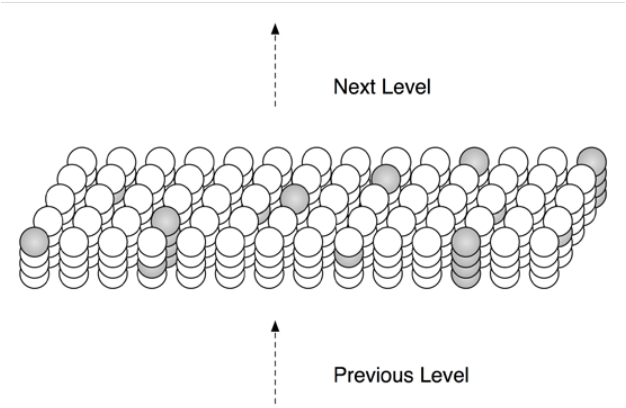


*Figure 2* By activating a subset of cells in each column, an HTM layer can represent the same input in many different contexts. Columns only activate predicted cells. Columns with no predicted cells activate all the cells in the column. The figure shows some columns with one cell active and some columns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

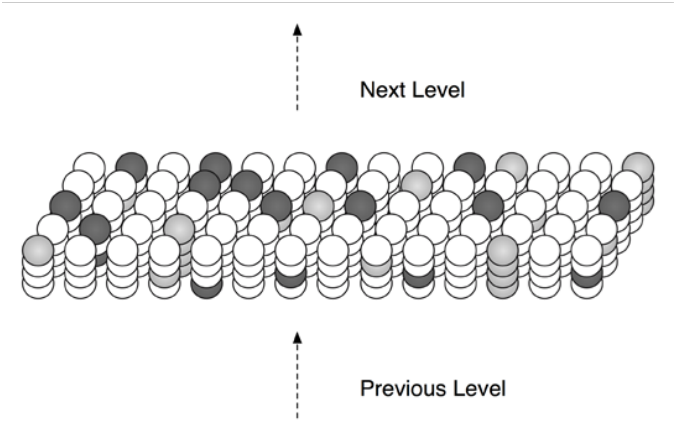


*Figure 3* At any point in time, some cells in an HTM layer will be active due to feed-forward input (shown in light gray). Other cells that receive lateral input from active cells will be in a predictive state (shown in dark gray).

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

The next step for the Temporal Memory algorithm 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.

## Temporal Memory Algorithm Pseudocode

The Temporal Memory algorithm starts where the Spatial Pooling algorithm leaves off, with a set of active columns representing the feed-forward input. In the pseudocode below, a time step consists of the following computations:

1. Receive a set of active columns, evaluate them against predictions, and choose a set of active cells:
    a. For each active column, check for cells in the column that have an active distal dendrite segment (i.e. cells that are in the “predictive state” from the previous time step), and activate them. If no cells have active segments, activate all the cells in the column, marking this column as “bursting”. The resulting set of active cells is the representation of the input in the context of prior input.
    b. For each active column, learn on at least one distal segment. For every bursting column, choose a segment that had some active synapses at any permanence level. If there is no such segment, grow a new segment on the cell with the fewest segments, breaking ties randomly. On each of these learning segments, increase the permanence on every active synapse, decrease the permanence on every inactive synapse, and grow new synapses to cells that were previously active.

2. Activate a set of dendrite segments: 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. The collection of cells with active distal dendrite segments will be the predicted cells in the next time step.

At the end of each time step, we increment the time step counter “t”.

Rather than storing a set of predicted cells, the algorithm stores a set of active distal dendritic segments. A cell is predicted if it has an active segment.

**1) Evaluate the active columns against predictions. Choose a set of active cells.**

Translate a set of active columns into a set of active cells, and grow / reinforce / punish distal synapses to better predict these columns.

First, for each column, decide whether it is any of the following:

- Active and has one or more predicted cells
- Active and has no predicted cells
- Inactive and has at least one predicted cell

Handle each case in a different function, defined further below.

```
1. for column in columns
2.     if column in activeColumns(t) then
3.         if count(segmentsForColumn(column, activeSegments(t-1))) > 0 then
4.             activatePredictedColumn(column)
5.         else
6.             burstColumn(column)
7.     else
8.         if count(segmentsForColumn(column, matchingSegments(t-1))) > 0 then
9.             punishPredictedColumn(column)
```

A column is predicted if any of its cells have an active distal dendrite segment (lines 3-4). If none of the column's cells have an active segment, the column bursts (line 6).

Dendrite segments in inactive columns are punished if they are active enough to be "matching" (lines 8-9).

```
10. function activatePredictedColumn(column)
11.     for segment in segmentsForColumn(column, activeSegments(t-1))
12.         activeCells(t).add(segment.cell)
13.         winnerCells(t).add(segment.cell)
14.
15.         if LEARNING_ENABLED:
16.             for synapse in segment.synapses
17.                 if synapse.presynapticCell in activeCells(t-1) then
18.                     synapse.permanence += PERMANENCE_INCREMENT
19.                 else
20.                     synapse.permanence -= PERMANENCE_DECREMENT
21.
22.             newSynapseCount = (SYNAPSE_SAMPLE_SIZE -
23.                                numActivePotentialSynapses(t-1, segment))
24.             growSynapses(segment, newSynapseCount)
```

For each active column, if any cell was predicted, those predicted cells become active (lines 11-12). Each of these cells is marked as a "winner" (line 13), making them presynaptic candidates for synapse growth in the next time step.

For each of these correctly active segments, reinforce the synapses that activated the segment, and punish the synapses that didn't contribute (lines 16-20). If the segment has fewer than SYNAPSE_SAMPLE_SIZE active synapses, grow new synapses to a subset of the winner cells from the

previous time step to make up the difference (lines 22 – 24).

```
25. function burstColumn(column)
26.     for cell in column.cells
27.         activeCells(t).add(cell)
28.
29.     if segmentsForColumn(column, matchingSegments(t-1)).length > 0 then
30.         learningSegment = bestMatchingSegment(column)
31.         winnerCell = learningSegment.cell
32.     else
33.         winnerCell = leastUsedCell(column)
34.         if LEARNING_ENABLED:
35.             learningSegment = growNewSegment(winnerCell)
36.
37.     winnerCells(t).add(winnerCell)
38.
39.     if LEARNING_ENABLED:
40.         for synapse in learningSegment.synapses
41.             if synapse.presynapticCell in activeCells(t-1) then
42.                 synapse.permanence += PERMANENCE_INCREMENT
43.             else
44.                 synapse.permanence -= PERMANENCE_DECREMENT
45.
46.         newSynapseCount = (SAMPLE_SIZE -
47.                            numActivePotentialSynapses(t-1, learningSegment))
48.         growSynapses(learningSegment, newSynapseCount)
```

If the column activation was unexpected, then each cell in the column becomes active (lines 26-27).

Select a winner cell and a learning segment for the column (lines 29-35). If any cells have a matching segment, select the best matching segment and its cell (lines 30-31). Otherwise select the least used cell and grow a new segment on it (line 33-35).

On the learning segment, reinforce the synapses that partially activated the segment, and punish the synapses that didn't contribute (lines 40-44). Then grow new synapses to a subset of the previous time step's winner cells (lines 46-48).

```
49. function punishPredictedColumn(column)
50.     if LEARNING_ENABLED:
51.         for segment in segmentsForColumn(column, matchingSegments(t-1))
52.             for synapse in segment.synapses
53.                 if synapse.presynapticCell in activeCells(t-1) then
54.                     synapse.permanence -= PREDICTED_DECREMENT
```

When a column with matching segments doesn't become active, punish the synapses that caused these segments to be "matching".

**2) Activate a set of dendrite segments.**

Calculate a set of active and matching dendrite segments. Active segments denote predicted cells. Matching segments are used in the next time step when choosing which cells to learn on.

```
55. for segment in segments
56.     numActiveConnected = 0
57.     numActivePotential = 0
58.     for synapse in segment.synapses
59.         if synapse.presynapticCell in activeCells(t) then
60.             if synapse.permanence ≥ CONNECTED_PERMANENCE then
61.                 numActiveConnected += 1
62.
63.             if synapse.permanence ≥ 0 then
64.                 numActivePotential += 1
65.
66.     if numActiveConnected ≥ ACTIVATION_THRESHOLD then
67.         activeSegments(t).add(segment)
68.
69.     if numActivePotential ≥ LEARNING_THRESHOLD then
70.         matchingSegments(t).add(segment)
71.
72.     numActivePotentialSynapses(t, segment) = numActivePotential
```

A synapse is “connected“ if its permanence is sufficiently high (line 60). For each segment, count the connected and potential synapses that are active (lines 58-64). If either number is high enough, the segment is considered “active” or “matching”, respectively (lines 66-70).

Record the number of active potential synapses so that we can use it during learning in the next time step (line 72, used in lines 23, 47).

Now, repeat. Go back to Stage 1. Putting these two stages together, you get the entire algorithm.

**Supporting functions**

These functions are used above.

```
73. function leastUsedCell(column)
74.     fewestSegments = INT_MAX
75.     for cell in column.cells
76.         fewestSegments = min(fewestSegments, cell.segments.length)
77.
78.     leastUsedCells = []
79.     for cell in column.cells
80.         if cell.segments.length == fewestSegments then
81.             leastUsedCells.add(cell)
82.
83.     return chooseRandom(leastUsedCells)
```

The least used cell is the cell with the fewest segments. Find all the cells with the fewest segments (lines 74-81) and choose one randomly (line 83).

```
84. function bestMatchingSegment(column)
85.     bestMatchingSegment = None
86.     bestScore = -1
87.     for segment in segmentsForColumn(column, matchingSegments(t-1))
88.         if numActivePotentialSynapses(t-1, segment) > bestScore then
89.             bestMatchingSegment = segment
90.             bestScore = numActivePotentialSynapses(t-1, segment)
91.
92.     return bestMatchingSegment
```

The best matching segment is the “matching” segment with largest number of active potential synapses.

```
93. function growSynapses(segment, newSynapseCount)
94.     candidates = copy(winnerCells(t-1))
95.     while candidates.length > 0 and newSynapseCount > 0
96.         presynapticCell = chooseRandom(candidates)
97.         candidates.remove(presynapticCell)
98.
99.         alreadyConnected = false
100.        for synapse in learningSegment.synapses
101.            if synapse.presynapticCell == presynapticCell then
102.                alreadyConnected = true
103.
104.        if alreadyConnected == false then
105.            newSynapse = createNewSynapse(segment, presynapticCell,
106.                                          INITIAL_PERMANENCE)
107.            newSynapseCount -= 1
```

When growing synapses, grow to a random subset of the previous time step’s winner cells. A segment can only connect to a presynaptic cell once (lines 99-104).

## Temporal Memory Implementation, Data Structures and Routines

In this section we describe some of the details of our Temporal Memory implementation and terminology.

Each Temporal Memory cell has a list of dendrite segments, where each segment contains a list of synapses. Each synapse has a presynaptic cell and a permanence value.

The implementation of potential synapses is different from the implementation in the Spatial Pooling algorithm. In the Spatial Pooling algorithm, the complete list of potential synapses is represented as an explicit list. In the Temporal Memory algorithm, 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 algorithm, we randomly add active synapses to each segment during learning (controlled by the parameter SYNAPSE_SAMPLE_SIZE). 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 Temporal Memory algorithm allows three mutually exclusive cell states: “active”, “inactive”, and “predictive”. But in the pseudocode above, it is possible for a cell to be both active and predictive. This happens occasionally, for example with the sequence “AABC”, if you burst “A”, the Temporal Memory algorithm will predict “A” and “B”, so at this point some “A” cells are both active and predictive.

The Temporal Memory algorithm allows a single distal dendrite segment to form connections to multiple patterns of previously active cells, but the pseudocode above generally does not do this. It learns one pattern per segment. This approach grows more dendrite segments than is strictly necessary, but this doesn’t have a fundamental impact on the algorithm.

| | |
|---|---|
| t | The current time step. This is incremented after the "Activate Dendrites" phase. |
| columns | A list of all columns |
| segments | A list of all segments |
| activeColumns(t) | Set of active columns at time t. This is the input to the Temporal Memory. |
| activeCells(t) | Set of active cells at time t |
| winnerCells(t) | Set of winner cells at time t |
| activeSegments(t) | Set of active segments at time t |
| matchingSegments(t) | Set of matching segments at time t |
| numActivePotentialSynapses (t, segment) | The number of active potential synapses on the given segment at time t. A "potential synapse" is a synapse with any permanence – it doesn't have to reach the CONNECTED_PERMANENCE. |
| ACTIVATION_THRESHOLD | Activation threshold for a segment. If the number of active connected synapses in a segment is ≥ this value, the segment is "active". |
| INITIAL_PERMANENCE | Initial permanence value for a synapse. |
| CONNECTED_PERMANENCE | If the permanence value for a synapse is ≥ this value, it is "connected". |
| LEARNING_THRESHOLD | Learning threshold for a segment. If the number of active potential synapses in a segment is ≥ this value, the segment is "matching", and it is qualified to grow and reinforce synapses to the previous active cells. |
| LEARNING_ENABLED | Either True or False. If True, the Temporal Memory should grow / reinforce / punish synapses each timestep. |
| PERMANENCE_INCREMENT | If a segment correctly predicted a cell's activity, the permanence values of its active synapses are incremented by this amount. |
| PERMANENCE_DECREMENT | If a segment correctly predicted a cell's activity, the permanence values of its inactive synapses are decremented by this amount. |
| PREDICTED_DECREMENT | If a segment incorrectly predicted a cell's activity, the permanence values of its active synapses are decremented by this amount. |
| SYNAPSE_SAMPLE_SIZE | The desired number of active synapses on a segment. A learning segment will grow synapses to reach this number. The segment connects to a subset of an SDR, and this is the size of that subset. |

Table 1. Variables and data structures used in the Temporal Memory pseudocode.

| | |
|---|---|
| segmentsForColumn(column, segments) | Given a set of segments, return the segments that are on a cell in this column. |
| count(collection) | Get the number of elements in this collection. |
| chooseRandom(collection) | Return a random element from this collection. |
| collection.add(element) | Add an element to this collection. Don't allow duplicates |
| collection.remove(element) | Remove an element from this collection. |
| chooseRandom(collection) | Return a random element from this collection. |
| min(a, b) | Equivalent to "if a ≤ b then return a else return b" |

Table 2. Supporting routines used in the Temporal Memory pseudocode. They may have different names in the NuPIC codebase.