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 Spatial Pooler (SP) 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.

Spatial Pooler Algorithm Implementation and Pseudocode

Spatial Pooler Algorithm Concepts

The most fundamental function of the spatial pooler is to convert a region's input into a sparse pattern. This function is important because the mechanism used to learn sequences and make predictions requires starting with sparse distributed patterns.

There are several overlapping goals for the spatial pooler, which determine how the spatial pooler operates and learns.

1) Use All Columns

An HTM layer has a fixed number of columns that learn to represent common patterns in the input. One objective is to make sure all the columns learn to represent something useful regardless of how many columns you have. We don't want columns that are never active. To prevent this from happening, we keep track of how often a column is active relative to its neighbors. If the relative activity of a column is too low, it boosts its input activity level until it starts to be part of the winning set of columns. In essence, all columns are competing with their neighbors to be a participant in representing input patterns. If a column is not very active, it will become more aggressive. When it does, other columns will be forced to modify their input and start representing slightly different input patterns.

2) Maintain Desired Density

A region needs to form a sparse representation of its inputs. Columns with the most input inhibit their neighbors. There is a radius of inhibition that is proportional to the size of the receptive fields of the columns (and therefore can range from small to the size of the entire region). Within the radius of inhibition, we allow only a percentage of the columns with the most active input to be "winners". The remainders of the columns are disabled. (A "radius" of inhibition implies a 2D arrangement of columns, but the concept can be adapted to other topologies.)

It there is no topology we use an infinite radius of inhibition. In this case a fixed percentage of the columns with the most active input become "winners" and inhibit the rest of the columns.

3) Avoid Trivial Patterns

We want all our columns to represent non-trivial patterns in the input. This goal can be achieved by setting a minimum threshold of input for the column to be active. For example, if we set the threshold to 10, it means that a column must have a least 10 active synapses on its dendrite segment to be active, guaranteeing a certain level of complexity to the pattern it represents.

4) Avoid Extra Connections

If we aren't careful, a column could form a large number of valid synapses. It would then respond strongly to many different unrelated input patterns. Different subsets of the synapses would respond to different patterns. To avoid this problem, we decrement the permanence value of any synapse that isn't currently contributing to a winning column. By making sure non-contributing synapses are sufficiently penalized, we guarantee a column represents a limited number input patterns, sometimes only one.

5) Self-adjusting Receptive Fields

Real brains are highly "plastic"; regions of the neocortex can learn to represent entirely different things in reaction to various changes. If part of the neocortex is damaged, other parts will adjust to represent what the damaged part used to represent. If a sensory organ is damaged or changed, the associated part of the neocortex will adjust to represent something else. The system is self-adjusting.

We want our HTM regions to exhibit the same flexibility. If we allocate 10,000 columns to a layer, it should learn how to best represent the input with 10,000 columns. If we allocate 20,000 columns, it should learn how best to use that number. If the input statistics change, the columns should change to best represent the new reality. In short, the designer of an HTM should be able to allocate any resources to a region and the region will do the best job it can of representing the input based on the available columns and input statistics. The general rule is that with more columns in a region, each column will represent larger and more detailed patterns in the input. Typically the columns also will be active less often, yet we will maintain a relatively constant density.

No new learning rules are required to achieve this highly desirable goal. By boosting inactive columns, inhibiting neighboring columns to maintain sparsity, establishing minimal thresholds for input, maintaining a large pool of potential synapses, and adding and forgetting synapses based on their contribution, the ensemble of columns will dynamically configure to achieve the desired effect.

Spatial Pooler Algorithm Details

We can now go through everything the spatial pooling function does.

1) Start with an input consisting of a fixed number of bits. These bits might represent sensory data or they might come from another layer elsewhere in the system.

2) Assign a fixed number of columns to the region receiving this input. Each column has an associated dendrite segment. Each dendrite segment has a set of potential synapses representing a subset of the input bits. Each potential synapse has a permanence value. Based on their permanence values, some of the potential synapses will be connected.

3) For any given input, determine how many connected synapses on each column are connected to active input bits.

4) The number of active synapses is multiplied by a "boosting" factor which is dynamically determined by how often a column is active relative to its neighbors.

5) A fixed number of columns within the inhibition radius with the highest activations after boosting become active and disable the rest of the columns within the radius. The inhibition radius is itself dynamically determined by the spread (or "fan-out") of input bits. There is now a sparse set of active columns.

6) For each of the active columns, we adjust the permanence values of all the potential synapses. The permanence values of synapses aligned with active input bits are increased. The permanence values of synapses aligned with inactive input bits are decreased. The changes made to permanence values may change some synapses from being connected to not connected, and vice-versa.

Spatial Pooler Pseudocode

This section contains the detailed pseudocode for the spatial pooler function. The input to this code is an array of bottom-up binary inputs from sensory data or the previous level. The code computes activeColumns(t) - the list of columns that win due to the bottom-up input at time t. This list is the output of the spatial pooling routine and is then sent as input to the temporal memory routine (see chapter on Temporal Memory).

After initialization, every iteration of the spatial pooler's compute routine goes through three distinct phases that occur in sequence:

Phase 1: compute the overlap with the current input for each column

Phase 2: compute the winning columns after inhibition

Phase 3: update synapse permanence and internal variables

Although spatial pooler learning is inherently online, you can turn off learning by simply skipping Phase 3.

The rest of the chapter contains the pseudocode for each of the three steps. The various data structures and supporting routines used in the code are defined at the end.

Initializing Spatial Pooler Parameters

Prior to receiving any inputs, the code is initialized by computing a list of initial potential synapses for each column. This consists of a random set of inputs selected from the input space. Each input is represented by a synapse and assigned a random permanence value. The random permanence values are chosen with two criteria. First, the values are chosen to be in a small range around connectedPerm (the minimum permanence value at which a synapse is considered "connected"). This enables potential synapses to become connected (or disconnected) after a small number of training iterations. Second, each column has a natural center over the input region, and the permanence values have a bias towards this center (they have higher values near the center).

Phase 1: Overlap

Given an input vector, the first phase calculates the overlap of each column with that vector. The overlap for each column is simply the number of connected synapses with active inputs, multiplied by its boost. If this value is below stimulusThreshold, we set the overlap score to zero.

```
for c in columns

  overlap(c) = 0
  for s in connectedSynapses(c)
      overlap(c) = overlap(c) + input(t, s.sourceInput)

  if overlap(c) < stimulusThreshold then
      overlap(c) = 0
  else
      overlap(c) = overlap(c) * boost(c)
```

Phase 2: Inhibition

The second phase calculates which columns remain as winners after the inhibition step. numActiveColumnsPerInhArea is a parameter that controls the number of columns that end up winning. For example, if numActiveColumnsPerInhArea is 10, a column will be a winner if its overlap score is greater than the score of the 10'th highest column within its inhibition radius.

```
11.  for c in columns
12.
13.      minLocalActivity = kthScore(neighbors(c), numActiveColumnsPerInhArea)
14.
15.      if overlap(c) > 0 and overlap(c) ≥ minLocalActivity then
16.          activeColumns(t).append(c)
17.
```

Phase 3: Learning

The third phase performs learning; it updates the permanence values of all synapses as necessary, as well as the boost and inhibition radius.

The main learning rule is implemented in lines 20-26. For winning columns, if a synapse is active, its permanence value is incremented, otherwise it is decremented. Permanence values are constrained to be between 0 and 1.

Lines 28-36 implement boosting. There are two separate boosting mechanisms in place to help a column learn connections. If a column does not win often enough (as measured by activeDutyCycle), its overall boost value is increased (line 30-32). Alternatively, if a column's connected synapses do not overlap well with any inputs often enough (as measured by overlapDutyCycle), its permanence values are boosted (line 34-36). Note: once learning is turned off, boost(c) is frozen.

Finally, at the end of Phase 3 the inhibition radius is recomputed (line 38).

```
18.  for c in activeColumns(t)
19.
20.      for s in potentialSynapses(c)
21.          if active(s) then
22.              s.permanence += synPermActiveInc
23.              s.permanence = min(1.0, s.permanence)
24.          else
25.              s.permanence -= synPermInactiveDec
26.              s.permanence = max(0.0, s.permanence)
27.
28.  for c in columns:
29.
30.      minDutyCycle(c) = 0.01 * maxDutyCycle(neighbors(c))
31.      activeDutyCycle(c) = updateActiveDutyCycle(c)
32.      boost(c) = boostFunction(activeDutyCycle(c), minDutyCycle(c))
33.
34       overlapDutyCycle(c) = updateOverlapDutyCycle(c).
35.      if overlapDutyCycle(c) < minDutyCycle(c) then
36.          increasePermanences(c, 0.1*connectedPerm)
37.
38.  inhibitionRadius = averageReceptiveFieldSize()
```

Spatial Pooler Data structures, Routines and Parameters

The following variables and data structures are used in the spatial pooler pseudocode:

columns	List of all columns.
columnCount	The total number of columns in the spatial pooler.
input(t,j)	The input to this level at time t. input(t, j) is 1 if the j'th input is on.
overlap(*c*)	The spatial pooler overlap of column c with a particular input pattern.
activeColumns(*t*)	List of column indices that are winners due to bottom-up input.
numActiveColumnsPerInhArea	A parameter controlling the number of columns that will be winners after the inhibition step.
inhibitionRadius	Average connected receptive field size of the columns.
neighbors(c)	A list of all the columns that are within inhibitionRadius of column c.
stimulusThreshold	A minimum number of inputs that must be active for a column to be considered during the inhibition step.
boost(c)	The boost value for column c as computed during learning - used to increase the overlap value for inactive columns.
maxBoost	The maximum value of boost(c).
synapse	A data structure representing a synapse - contains a permanence value and the source input index.
potentialPct	The percent of the inputs, within a column's potential radius, that are initialized to be in this column's potential synapses.
connectedPerm	If the permanence value for a synapse is greater than this value, it is said to be connected.
potentialSynapses(c)	The list of potential synapses and their permanence values for this column.
connectedSynapses(c)	A subset of potentialSynapses(c) where the permanence value is greater than connectedPerm. These are the bottom-up inputs that are currently connected to column c.
synPermActiveInc	Amount permanence values of active synapses are incremented during learning.
synPermInactiveDec	Amount permanence values of inactive synapses are decremented during learning.

activeDutyCycle(c)	A sliding average representing how often column c has been active after inhibition (e.g. over the last 1000 iterations).
overlapDutyCycle(c)	A sliding average representing how often column c has had significant overlap (i.e. greater than stimulusThreshold) with its inputs (e.g. over the last 1000 iterations).
minDutyCycle(c)	A variable representing the minimum desired firing rate for a cell. If a cell's firing rate falls below this value, it will be boosted. This value is calculated as 1% of the maximum firing rate of its neighbors.

The following supporting routines are used in the above code.

kthScore(cols, k)	Given the list of columns, return the k'th highest overlap value.
updateActiveDutyCycle(c)	Computes a moving average of how often column c has been active after inhibition.
updateOverlapDutyCycle(c)	Computes a moving average of how often column c has overlap greater than stimulusThreshold.
averageReceptiveFieldSize()	The radius of the average connected receptive field size of all the columns. The connected receptive field size of a column includes only the connected synapses (those with permanence values >= connectedPerm). This is used to determine the extent of lateral inhibition between columns.
maxDutyCycle(cols)	Returns the maximum active duty cycle of the columns in the given list of columns.
active(s)	True if synapse s is active, i.e. the input connected to synapse s is ON.
increasePermanences(c, s)	Increase the permanence value of every synapse in column c by a scale factor s.
boostFunction(c)	Returns the boost value of a column. The boost value is a scalar between 1 and maxBoost. If activeDutyCyle(c) is above minDutyCycle(c), the boost value is 1. The boost increases linearly once the column's activeDutyCyle starts falling below its minDutyCycle up to a maximum value maxBoost.

We have found that the following parameter settings work well under a wide range of scenarios:

columnCount	This is task dependent but we recommend a minimum value of 2048.
numActiveColumnsPerInhArea	We usually set this to be 2% of the expected inhibition radius. For 2048 columns and global inhibition, this is set to 40. We recommend a minimum value of 25.
stimulusThreshold	This is roughly the background noise level expected out of the encoder and is often set to a very low value (0 to 5). The system is not very sensitive to this parameter; set to 0 if unsure.
maxBoost	Boosting is only required for scenarios with complex set of inputs (such as vision), so maxBoost is often set to 1.0. For applications that require boosting, values of 2.0 to 10.0 are used.
potentialPct	This should be set so that on average, at least 15-20 input bits are connected when the spatial pooler is initialized. For example, suppose the input to a column typically contains 40 ON bits and that permanences are initialized such that 50% of the synapses are initially connected. In this case you will want potentialPct to be at least 0.75 since 40*0.5*0.75 = 15.
connectedPerm	This is usually set to 0.2. The spatial pooler is not sensitive to this parameter.
synPermActiveInc	This parameter is somewhat data dependent. (The amount of noise in the data will determine the optimal ratio between synPermActiveInc and synPermInactiveDec.) Usually set to a small value, such as 0.03
synPermInactiveDec	Usually set to a smaller value, such as 0.015