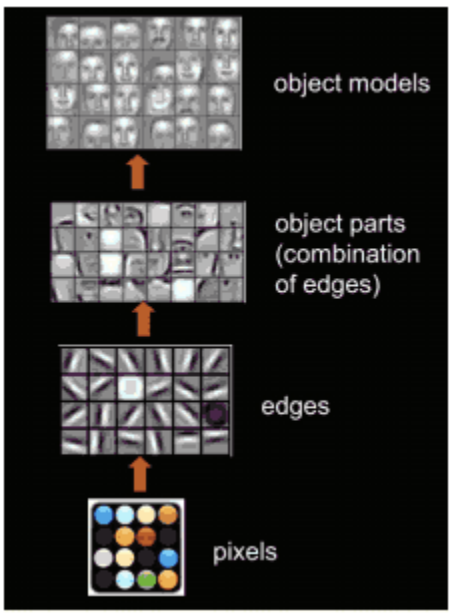
Introduction to the CLA Algorithm

Hierarchies, sparseness and the Google's cat

When I started my studies on brain inspired machine learning algorithms, trying to understand the famous Google's cat achievement, I came across to the concepts of hierarchy and sparseness. Back to 1958, David Hubble and Torsten Wiesel found cells specialized in identifying oriented lines in the visual cortex. These cells fire when they detect an edge oriented at a particular angle, in a specific area of the retina. One question arises: why lines? Many researches agreed that this type of pattern was reoccuring frequently in natural visual scenes, which might justify why the brain would have an explicit representation of such patterns. Using a **sparse** set of lines in several different orientations, the visual cortex



can make a first reconstruction (maybe not perfect) of the image that is coming from the retina. It would be desirable to have a good representation of the visual data while reducing neuron firing to a minimum. Probably the brain makes some simplifications in favor of energy economy and speed. The specialized cells do this trick. Maybe, the same optimization occurs in all layers of the visual cortex. Moreover, maybe the same learning algorithm is used in many regions of the brain to discover the building blocks of speech, sound, touch, language, etc. The picture beside exemplifies how this possibly happen. In the first layer, the visual cortex select the cells that represent the best match between the image and the set of edges available. In the second layer, are selected cells that represent combinations of the edges chosen in the first layer. And so on. The world is made by **hierarchies** and the brain should take advantage of that.

The schema described above was the inspiration for the new researches in Machine Learning algorithms, more specifically, in artificial neural networks and the **deep learning architectures**. In deep learning, the idea is to come up with a basis, a pool of shared building blocks, so that every data instance can be reconstructed as a different linear combination of the same building blocks. In a two-layer deep learning algorithm, the data is in layer 0 and the building blocks are in layer 1. Now treat the layer-1 building blocks as your new data instances and try to find layer-2 building blocks to reconstruct the layer-1 building blocks. You now have a three-layer deep learning model. Pile on a few more layers and you end up with a nice deep representation model, in which the data lies at the bottom, and every layer maintains only the essential ingredients required to reconstruct the patterns in the layer beneath it. Once the whole model is trained, you can feed it with a data instance and it will find the activation levels (the coefficients of the linear combinations) of all the building blocks in all the layers. These activation levels form a new representation of the picture, no longer in the language of pixels but in terms of a weighted sum of the different building

blocks. The hope is that once you learn the model using a dataset like Caltech 101, the building blocks at higher levels will be activated only when a specific type of object appears in the image, for example a cat or a dog. Hence by looking at these high-level building blocks, one will be able to tell which type of object appears in the picture.

I decided to start the text with this introduction to elucidate a misunderstanding I made when i began studying the CLA theory. Although CLA is also strongly influenced by new discoveries about the brain, the concepts of sparseness and hierarchies described until now are not the same as in CLA, as we will soon see in the next sessions.

The CLA Algorithm

The CLA - Cortical Learning Algorithm aims to simulate a region in the neocortex. It is a collection of cells in a tabular organisation. Cells establish connections with others cells via synapses. The connections within cells in the same column are made via the proximal dendrite and connections within cells laterally via the distal dendrites. Figure 1 depict this schema.

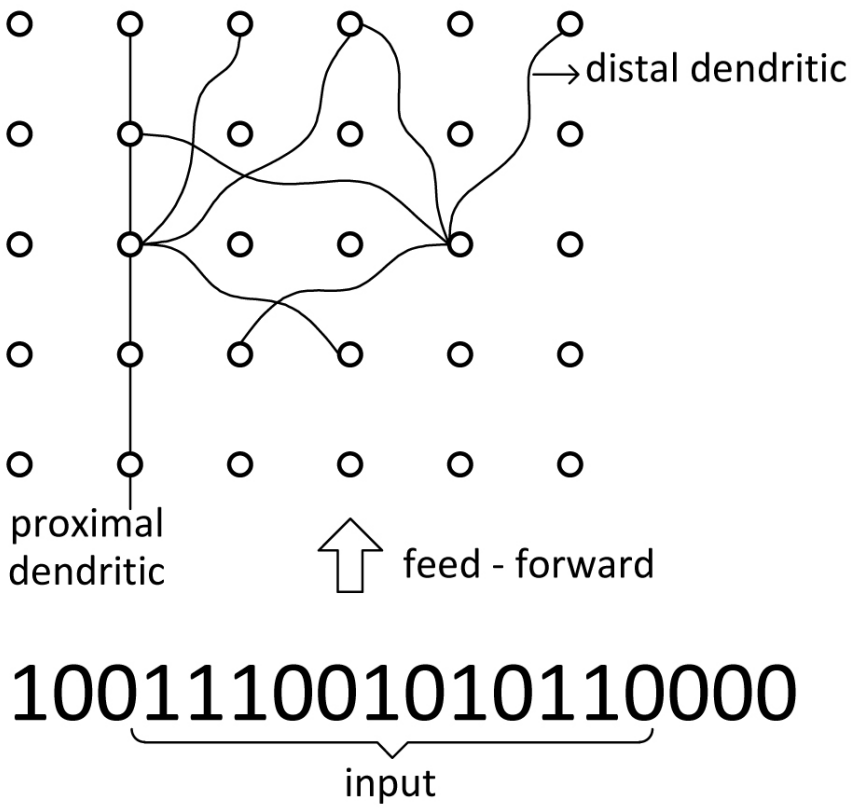
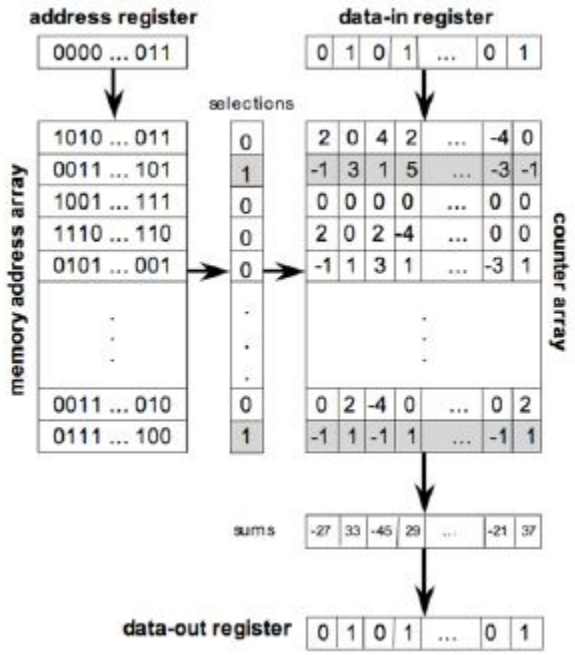


Figure 1 - General CLA region

The objective of a region is to receive an input of bits (usually large) and learn how to identify patterns that occur frequently enough. More than that, also learn to identify which order they occur. If a pattern representing the letter A occurs, following a pattern representing the letter B occurs several times, than the region learns what are the patterns that represent A and B

and learns to predict that after A comes B. In the CLA algorithm, the structure responsible to detect patterns is the Spatial Pooler - SP, and the structure responsible to learn sequences of patterns is the Temporal Pooler - TP.

In the heart of the technology is the theory of Sparse Distributed Representation - SDR. SDR is related to the work of Sparse Distributed Memory made by P. Kanerva in 1989. He proposed a schema to built a specialized hardware that is an associative memory and emulates, he claimed, how the long term memory of the brain works. The main motivation is that the cerebro have tons of things to remember. So, it does that only with things that occur frequently enough. Moreover, it stores things that happen near in time all together, maintaining the context. In Kanerva's model, sensory input is represented in the form of very long bit vectors containing thousands or tens of thousands of bits. Because no two external situations are identical, the memory must respond to partial matches between the current sensory pattern and previously stored patterns. The measure of dissimilarity between patterns is the number of bits in which they differ, a metric known as the Hamming distance. For example, the distance between 01101 and 10111 is 3 bits. When a new information arrives, it is stored in the buckets of memory that best matches it (some threshold based on the hamming distance), in a way where similar memories are stored in the same place. As the number of buckets are much less than the number of different inputs, we call them sparse. The image describes the overall model. This is a schema that sacrifices precision in benefit of space economy, but everything suggest that the brain works this way.



This schematic diagram shows the relations among the components of sparse distributed memory. The memory in this example stores and retrieves 256-bit patterns across 2,000 physical locations. Each horizontal row is a location. The input pattern (cue) in the address register is compared simultaneously to all 2,000 patterns in the memory address array; each line in the array holds the address of one location. The distances from each address pattern are compared with the memory's built-in threshold radius and a subset of the locations is selected (*shaded areas*). The 256-bit data pattern is stored at the selected locations by adding 1 to each counter in the counter array corresponding to each 1 in the pattern and subtracting 1 from each counter corresponding to a 0 in the pattern. A 256-bit pattern is retrieved by forming 256 sums from the corresponding counters in each selected location and then forming a 1 output bit in the data-out register for each sum that is nonnegative and a 0 for each sum that is negative. The retrieved pattern is a statistical reconstruction determined from the contents of all selected locations. All selections can be done in parallel, and all data bits can be handled in parallel, giving the memory great speed over a wide range of pattern widths and physical locations.

CLA highly depends on SDR to work properly. First of all, the input must be SDR compatible. It means that change in one bit doesn't affect much the whole pattern. Images are good examples of inputs that are naturally SDR compatible. Suppose the following set of bits representing the image of a dog eating a bone.

10011001010001010010000000001111000001101011101010001010000101010100001010100
01100010101111110100000000011100001101010000000011010000001110101010100000

00000000000100101000010101000011100000000000000001010101001100001011111010010100000

If some bits are changed from on to off, it will be not enough to change the meaning of the information. As Kanervas' studies suggest, in large inputs of this nature, something around of 50% of the bits are necessary to make connections between patterns that seemingly have little to do with each other. Keeping track of these bits you can identify the big pattern. If you see other input with the same bits on in the same position, you will have high probability that they are the same input, or, at least, highly related.

CLA needs inputs be SDR compatible to form a sparse representation model of the knowledge. CLA is designed to process lots of information as the brain does and memory efficiency is crucial.

Spatial Pooler

In the beginning, the region is an empty structure with no information. Each column has several potential synapses with a randomly chosen subset of input bits. The default is 50% of the input bits but could be more. We say "potential" because the synapse can be connected or not. Each synapse has a permanence value (varying from 0 to 1) that controls if the synapse is, in fact, connected. In simple terms, a synapse will be connected if it is related to a bit that is on (its value is 1) most of the time. As we saw, in the beginning the system has no information, so the first step of the SP algorithm is randomly initialize the synapses of the proximal dendrites with permanence values. The result is something like the Figure 2.

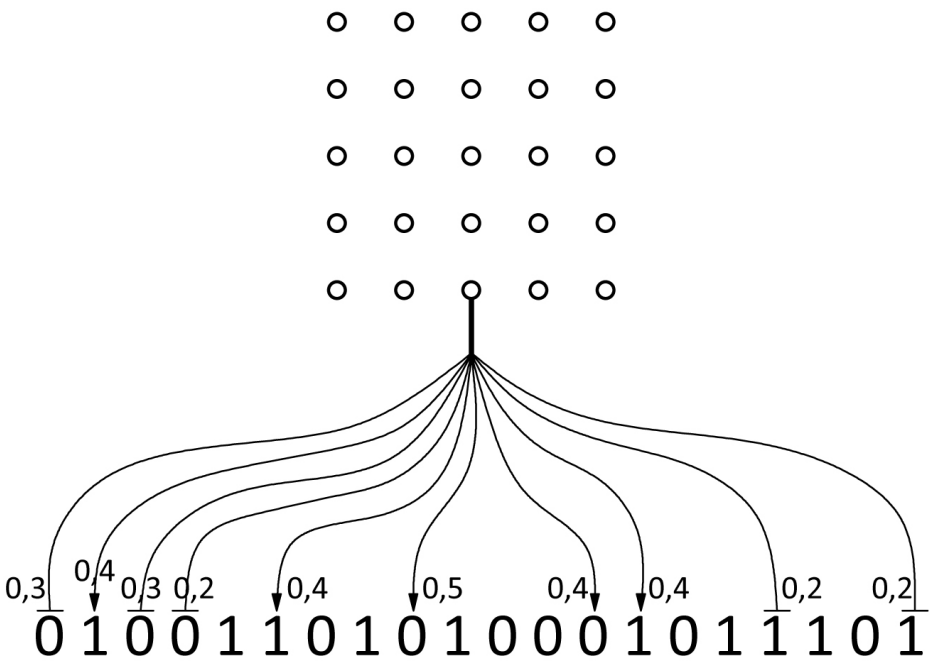


Figure 2 - Synapses and their permanence values

This will be useful for the region starts the learning process from some point. As inputs are arriving, in the feed-forwarding process, synapses are being connected and disconnected in

the attempt to represent the patterns. Let's see in detail how this occur. For each column, calculates the **overlap** for the column, i e, the number of **active synapses** that are connected with on bits. A synapse is connected if its **permanence value** is above a threshold (**connectedPerm**). If the value of the overlap is big enough (greater than **minOverlap**) the value is considered valid, otherwise is setted to zero. Consider the example in Figure 3. For a connectedPerm setted to 0.4 to indicate a connected synapse, the overlap for column 1 are the cells D and E. There is no connected synapse with an active bit for column 2, so its overlap is zero. The pseudo code below implements this logic.

```
1. for c in columns
2.
3.   overlap(c) = 0
4.   for s in connectedSynapses(c)
5.     overlap(c) = overlap(c) + input(t, s.sourceInput)
6.
7.   if overlap(c) < minOverlap then
8.     overlap(c) = 0
9.   else
10.   overlap(c) = overlap(c) * boost(c)
```

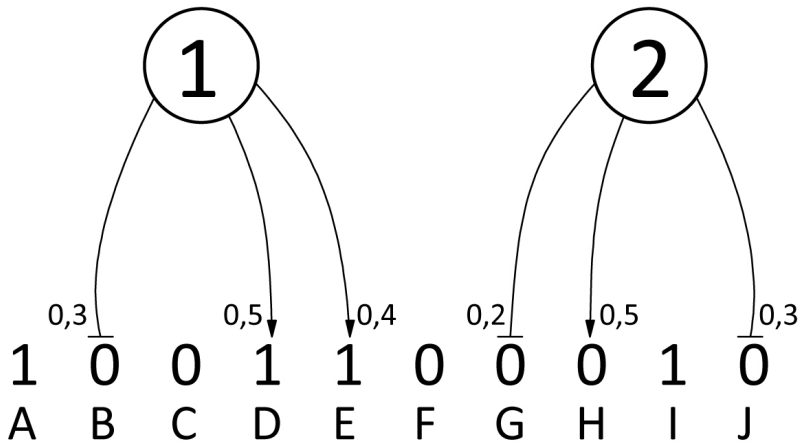


Figure 3 - Two columns and their potential synapses

The overlap value of a column is useful to decide whether a **column is active**. This is an important concept. The main objective of SP is, after each input that arrives, output a **sparse** set of columns that best fit the input bits. These are the selected active columns and they will be sparse because they are a tiny fraction of the total number of columns available. So, in the end, we have a sparse representation of the large input, because with few columns, totaling a small number of different bits, we can detect the large pattern.

The next pseudo code does the selection of the active columns.

```
11. for c in columns
12.
13.   minLocalActivity = kthScore(neighbors(c), desiredLocalActivity)
```

```
14.
15.   if overlap(c) > 0 and overlap(c) ≥ minLocalActivity then
16.     activeColumns(t).append(c)
17.
```

The method kthScore(neighbors(c), desiredLocalActivity) implements something called **inhibition**. Inhibition needs to be taken into account if we are considering the **topology** of the input. Topology in the SP relates to how each column position is connected with its input bits. Recall that each column is connected with a random subset of the input bits. But it is true only with no concern with topology. Depending the nature of the input, topology needs to be considered to achieve better results. For example, a SP learning to recognise features in a SDR coming in from a retina, would have columns looking at a set of input bits directly below it in the retina. The first column will be connected to the first n bits (or some 2D portion in the beginning of the image), the second will be connected to the adjacent bits and so on. In this case, local activity and the job of inhibition makes sense to guarantee sparsity and distribution. On the other hand, if no topology exist, there is no difference in use inhibition or not. Will be ok in choosing the columns with the highest overlap values to be the winning columns because each of them will be looking an aleatory sub set of bits spreaded all over the input and the resulting columns will have a good distribution.

So, the role of inhibition is to control how many columns will be, in fact, returned as active, making sure to yield a good sparse and distributed representation. Look the example in Figure 4.

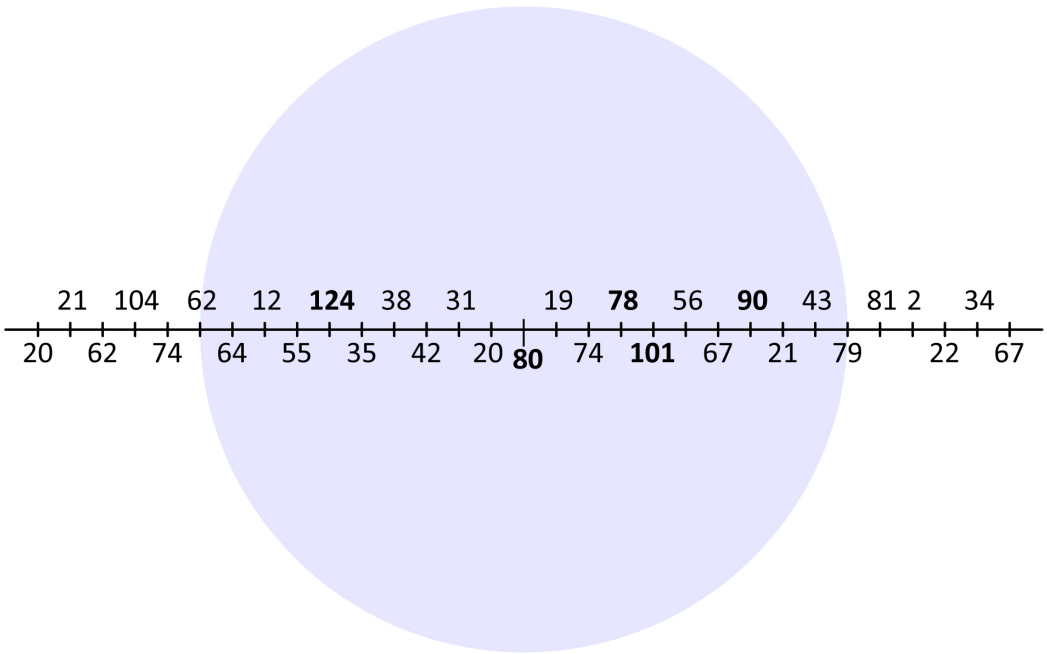


Figure 4 - Example of inhibition in action

When column 80 (a column whose overvalp value is 80) is evaluated, it is necessary to check its neighborhood. Suppose desiredLocalActivity is set to 5 and *inhibition radius* is 10. It

means that column 80 will be considered locally active if its overlap value is one of the fifth biggest compared to 10 (radius) columns left and right of it. In this case, the five biggest numbers within the radius are: 124, 101, 90, 80, 78. So, 80 would returned as active. Inhibition makes sure that within an area, say an area of 100 columns, only a small percentage (eg. 10) are active. The other columns are forced to stay inactive.

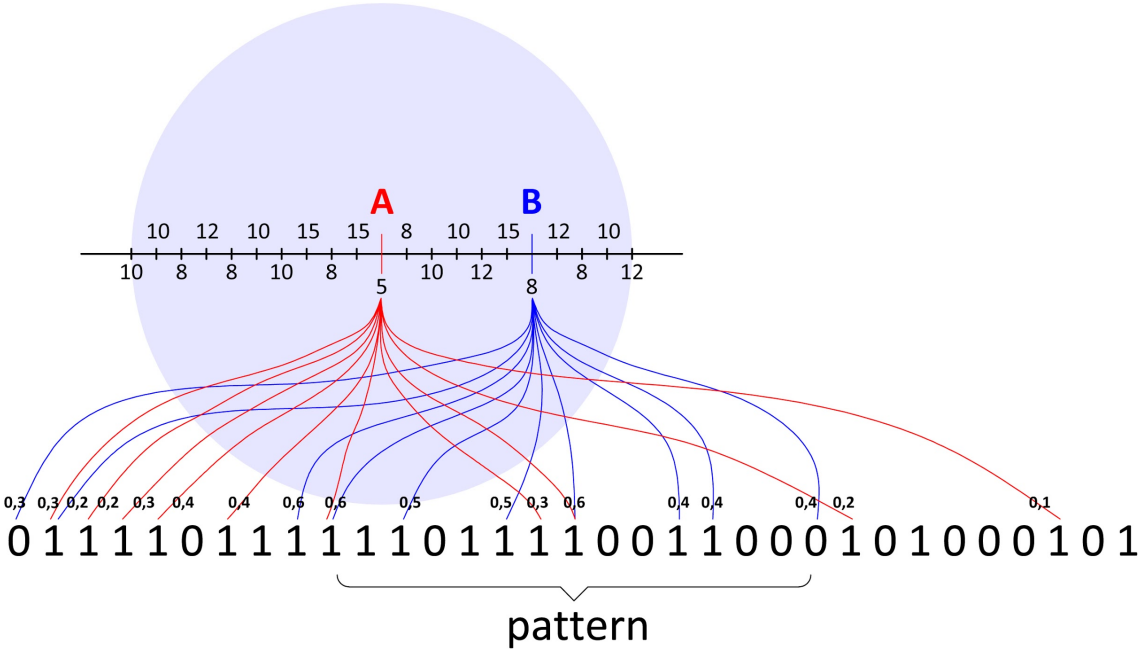


Figure 5 - a problem that inhibition can eventually correct

Other advantage of inhibition is that it can fix a problem that can arise on behalf of the random initialization of the permanence values. Consider Figure 5. Given a connectPerm of 0.4 (the threshold to a synapse be considered active), column A has 4 overlaps and column B has 8. But, if we look closer, we will see that column A is connected with 10 bits on, while column B with only 8. As a side effect of random initialization, column B beats column A even though column A has more connections with active bits of the pattern. In some cases (as the one depicted in the figure) inhibition may give a chance to these unfortunate columns by picking them as active.

Once the active columns have been chosen, the next step is to do the **learning**. Only for the active columns, increment the permanence value of the synapses connected to bits on and decrement otherwise.

```
18. for c in activeColumns(t)
19.
20.  for s in potentialSynapses(c)
21.    if active(s) then
22.      s.permanence += permanenceInc
23.      s.permanence = min(1.0, s.permanence)
24.    else
```

```
25.     s.permanence -= permanenceDec
26.     s.permanence = max(0.0, s.permanence)
27.
```

As an example of the effect of learning, Figure 6 shows how the synapses of Figure 3 were updated after the SP receive the same input twice.

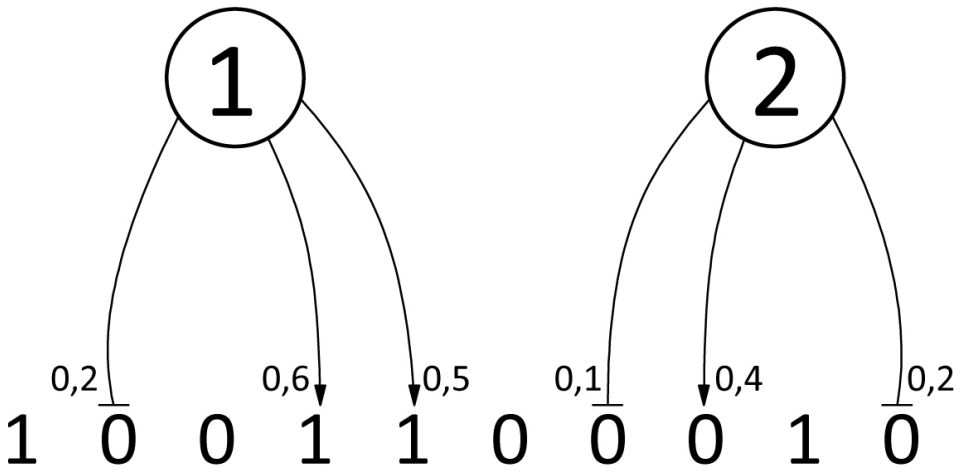


Figure 6 - Synapses being updated in the learning process

Doing all these steps repeatedly, one expected that a CLA region will gradually adjusting the synapses to recognise the most relevant inputs (the inputs that are more frequent). Bit by bit, the columns are becoming specialized to detect a relevant pattern. A combination of columns can arise a lot of different patterns representations.

Temporal Pooler

As we saw in Figure 1, a CLA region has columns of cells. The existence of many cells per column is necessary to temporal pooler do its job. One of the greatest ability of CLA technology is to make **predictions based on the context**. For example, consider the sequences of letters

BAZ
CAW

A CLA region is a **memory system** that learns to recognize that A occurs after B with some frequency (probability) and the same for the sequence CA. In other words, it knows that A can occur in the context of B or C. The TP uses the columns to represent such things. Let's see how. Observe Figure 7. In time T, two columns were active to represent the B pattern. The cells in these two columns sent stimulus to others cells nearby via **distal dendrites segments**. Other cells in the region will receive the stimulus and, if they are strong enough, the cell can enter the **predicted state**. In the figure, the cells highlighted are the cells in the predicted state in time T. In time T + 1, some other set of columns will be active. If in any of

these columns exist some cell in the predicted state, this cell is passed to the **active state**

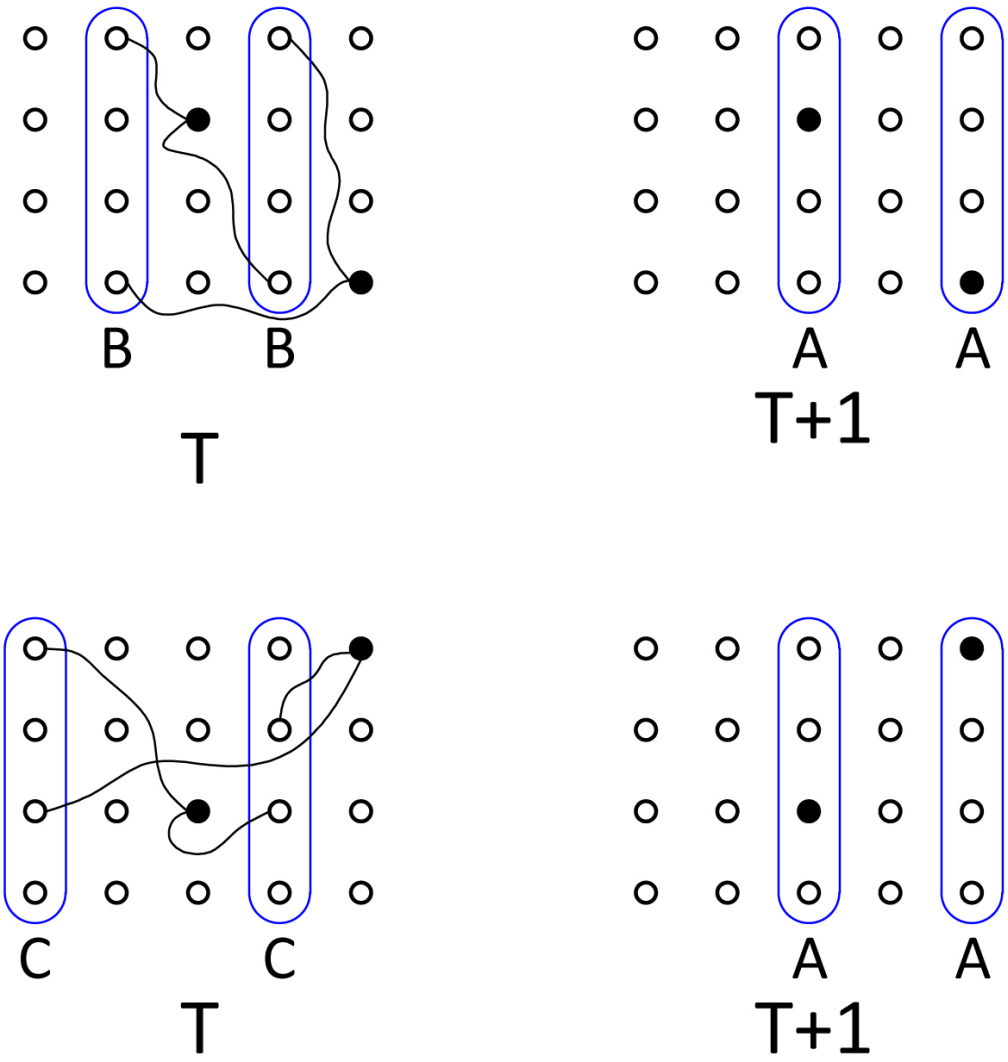


Figure 7 - Temporal Pooler in action

and its connections via distal dendrites with the columns that were active in time T are reinforced positively. That was what occurred in the example. The cells highlighted were in the predicted state in time T and after, when the prevision have been confirmed in time T + 1 and its columns become active, they entered the active state.

Note the representation of the A patterns in time T + 1. The same two columns are active but the cells are different. The representation of A coming from B is different from the representation coming from C. For different contexts we have different representation for the same pattern A. Why this is useful? Consider the next character in the sequence. They are different in the context of A coming from B than the A coming from C. So, as the cells that are active when the pattern A occurs are different, they can predicted distinct things. The two cells on top can predict Z and the other two predict W. In the end, the system naturally learned two forms to represent the sequences BAZ and CAW.

Now let's see in more details how all this mechanism works. Take a look in Figure 8. A cell

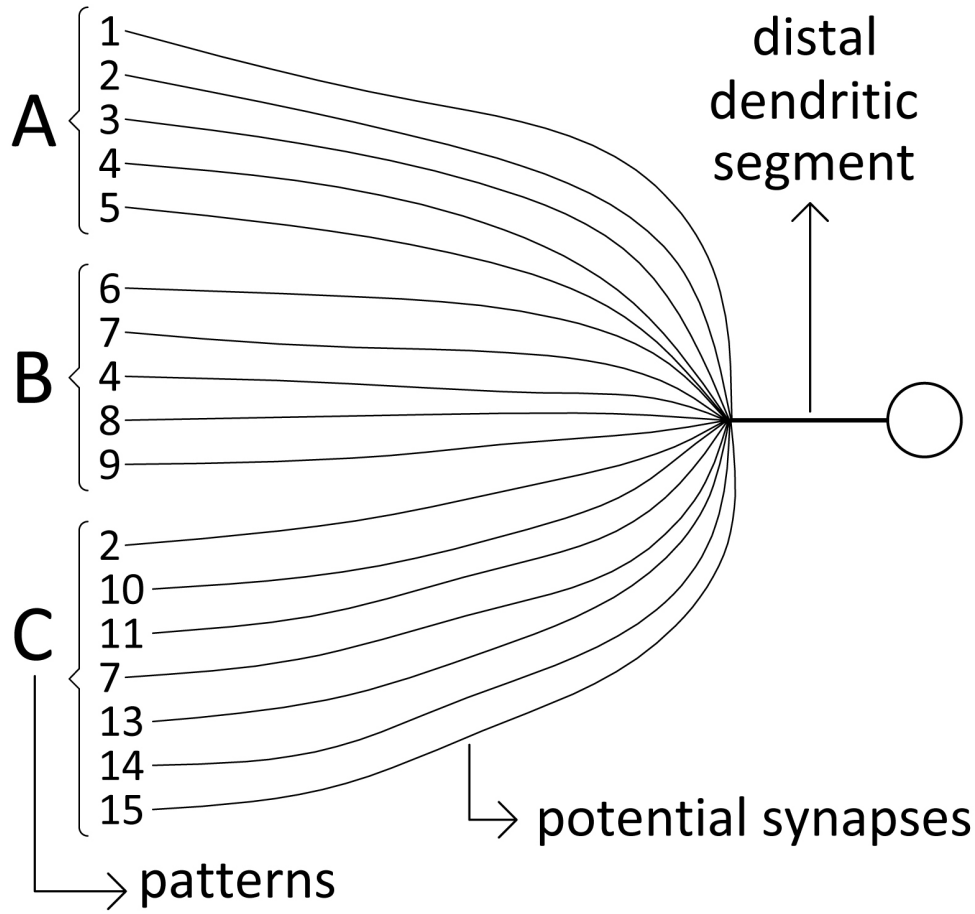


Figure 8 - A cell and its distal dendrite segment

can have one or more distal dendrite segment. As we saw, the segment is used by the cell to predict when it will be active in the future. In general, a segment has dozens potential synapses with others cells in the region. Here the concept of synapses are similar as in spatial pooler. Each synapse has a permanence value that will control whether the synapse is active or not. When a synapse is connected with an active cell, it is reinforced positively, increasing its permanence value and the chance of being connected. If a segment has enough active synapses (above some threshold) in some moment, than the **segment become active** and the cell enters the predicted state. A segment can learn several transitions. In Figure 8, the cell has five synapses specialized in detect when the pattern A occurs. These five synapses are binded with cells that are in the columns 1, 2, 3, 4 and 5. It means that when pattern A occurs via the same cells, the segment will become active (supposing five connected synapses are enough) and the cell will enter the predicted state. The others synapses formed more two predictions.

The code below receive the output of the spatial pooler algorithm, i. e., an array of zeros and ones flagging the positions where there is an active column.

1. for c in activeColumns(t)
2.

```
3.  buPredicted = false
4.  lcChosen = false
5.  for i = 0 to cellsPerColumn - 1
6.    if predictiveState(c, i, t-1) == true then
7.      s = getActiveSegment(c, i, t-1, activeState)
8.      if s.sequenceSegment == true then
9.        buPredicted = true
10.       activeState(c, i, t) = 1
11.       if segmentActive(s, t-1, learnState) then
12.           lcChosen = true
13.           learnState(c, i, t) = 1
14.
15.  if buPredicted == false then
16.    for i = 0 to cellsPerColumn - 1
17.      activeState(c, i, t) = 1
18.
19.  if lcChosen == false then
20.    I,s = getBestMatchingCell(c, t-1)
21.    learnState(c, i, t) = 1
22.    sUpdate = getSegmentActiveSynapses (c, i, s, t-1, true)
23.    sUpdate.sequenceSegment = true
24.    segmentUpdateList.add(sUpdate)
```

First, the cells that should be active in each active column are selected. The vector predictiveState(c, i, t-1) returns whether a cell i, in column c was predicted in time t-1. If true, the segment that was active in time t-1 is retrieved by the method getActiveSegment(c, i, t-1, activeState). If this is a “sequence segment”, i. e., this is a segment that is predicting the next step in the future (we will see that can exist segments pointing to more steps in the past), then set the cell as active (activeState(c, i, t) = 1). If the CLA region is in a learning state, flag the cell as in a learning state too. This is what happens until line 13.

In line 15 occurs what is called “bursting”. It is possible that in an active column returned by the SP there is no cell in the predicted state. So, in this case, all the cells in the column are put in the active state. No cell in the predicted state in time T means that in time T - 1 no active segment could be found in any cell of the column. Namely, when the segments of each cell of the column was verified, there weren’t the enough number of active synapses in none. To remedy this situation, the algorithm choose the segment that best match the configuration in time T - 1. This is done by the method getBestMatchingCell(c, t-1). For the given column c 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. When a cell is chosen, it is passed to the learn state.

Note that in the previous case, even with all the cells in the column in the active state, only one will be in the learn state. It is an important trick. Throughout the process of choose an

active distal dendrite segment, only synapses connected to cells in the learn state are considered. This avoids overrepresenting a fully active column in dendritic segments.

```
25.
26. for c, i in cells
27.   for s in segments(c, i)
28.     if segmentActive(s, t, activeState) then
29.       predictiveState(c, i, t) = 1
30.
31.       activeUpdate = getSegmentActiveSynapses (c, i, s, t, false)
32.       segmentUpdateList.add(activeUpdate)
33.
34.       predSegment = getBestMatchingSegment(c, i, t-1)
35.       predUpdate = getSegmentActiveSynapses( c, i, predSegment, t-1, true)
36.       segmentUpdateList.add(predUpdate)
```

The next step choose the cells that will enter the predicted state. Remember, we are in time T and have to consider the cells that entered the predicted state in time T - 1 and set the cells that will be evaluated in time T + 1. In line 31 are chosen the segments that best match the active columns in time T. In line 34 are chosen the segments that best match the active columns that were active in time T - 1. This will be a new segment added to the cell and will be used to predicted it further in the past.

```
37. for c, i in cells
38.   if learnState(s, i, t) == 1 then
39.     adaptSegments (segmentUpdateList(c, i), true)
40.     segmentUpdateList(c, i).delete()
41.   else if predictiveState(c, i, t) == 0 and predictiveState(c, i, t-1)==1 then
42.     adaptSegments (segmentUpdateList(c,i), false)
43.     segmentUpdateList(c, i).delete()
44.
```

The last step is the learning phase. In line 38 it updates the distal dendritic segments of cells that are in the learning state. Remember, to a cell be in the learning state in time T it was predicted in time T - 1 and the prediction was confirmed as its column become active in time T. So, the synapses of the segments are reinforced positively (adaptSegments (segmentUpdateList(c, i), true)). In line 41 the algorithm takes care of those cells that were predicted in time T- 1 but the prediction doesn’t occur. These are the cells that are not in the learning state and do not put again in the predicted state in time T.