Lecture 10

Unsupervised Learning NN: Hierarchical Temporal Memory - III

Implementation of SP and TP
Spatial Pooler Implementation

- **Input**
  - An array of bottom-up binary inputs from sensory data or the previous level

- **Output**
  - `activeColumns(t)`:
    - The list of columns that win due to the bottom-up input at time \( t \)
    - This list is then sent as input to the TP routine.

- **The pseudocode** consists of the following three phases:
  - Phase 1: Compute the overlap for each column
  - Phase 2: Compute the winning columns after inhibition
  - Phase 3: Update synapse permanence and internal variables
Initialization

- Prior to receiving any inputs, the region 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.
- Assign a random PV to the synapse
  - Chosen to be in a small range around connectedPerm (The minimum PV at which a synapse is considered "connected").
  - Each column has a natural center over the input region, and the PVs have higher values near the center.
Phase 1: Overlap

- Calculates the overlap of each column with the input vector

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

- Calculates which columns remain as winners after the inhibition step

```
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.
```
Phase 3: Learning

- Updates the PVs of all synapses, as well as the boost and inhibition radius

```python
18. for c in activeColumns(t)
19.     for s in potentialSynapses(c)
20.         if active(s) then
21.             s.permanence += permanenceInc
22.             s.permanence = min(1.0, s.permanence)
23.         else
24.             s.permanence -= permanenceDec
25.             s.permanence = max(0.0, s.permanence)
26.     for c in columns:
27.         minDutyCycle(c) = 0.01 * maxDutyCycle(neighbors(c))
28.         activeDutyCycle(c) = updateActiveDutyCycle(c)
29.         boost(c) = boostFunction(activeDutyCycle(c), minDutyCycle(c))
30.         overlapDutyCycle(c) = updateOverlapDutyCycle(c)
31.         if overlapDutyCycle(c) < minDutyCycle(c) then
32.             increasePermanences(c, 0.1*connectedPerm)
33.         inhibitionRadius = averageReceptiveFieldSize()
34.         inhibitionRadius = 0.5 * inhibitionRadius
35.         if inhibitionRadius < minDutyCycle(c) then
36.             decreasePermanences(c, 0.1*connectedPerm)
37.         else
38.             increasePermanences(c, 0.1*connectedPerm)
39.
```
TP computes the active and predictive state for each cell at $t$.

Input
- activeColumns($t$), as computed by the SP

Output
- The boolean OR of the active and predictive states for each cell for the next level.

The pseudocode:
- Phase 1: Compute the active state, activeState($t$), for each cell
- Phase 2: Compute the predicted state, predictiveState($t$), for each cell
- Phase 3: Update synapses
Cells maintain a list of dendrite segments
- Each segment contains a list of synapses plus a PV for each synapse.
- Changes to a cell's synapses are marked as temporary until the cell becomes active from FF input.
  - These temporary changes are maintained in segmentUpdateList.
- Each segment also maintains a boolean flag, sequenceSegment, indicating if the segment predicts FF input on the next time step.

The implementation of PSs is different from that in the SP.
- In the SP, the complete list of PSs is represented as an explicit list.
- In the TP, each segment can have its own (possibly large) list of PSs.
  - We randomly add active synapses to each segment during learning.
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 over-representing a fully active column in dendritic segments).
Phase 1: Compute the active state, $activeState(t)$, for each cell

- The first phase calculates the active state for each cell.
  - For each winning column, determine which cells should become active.
- For those columns, the code further selects one cell per column as the ‘learning cell’.
Temporal Pooler Implementation

- If the bottom-up input was predicted by any cell (i.e. its predictiveState output was 1 due to a sequence segment), then those cells become active (lines 23-27).

- If that segment became active from cells chosen with learnState on, this cell is selected as the learning cell (lines 28-30).

- If the bottom-up input was not predicted, then all cells in the column become active (lines 32-34).

- In addition, the best matching cell is chosen as the learning cell (lines 36-41) and a new segment is added to that cell.
Temporal Pooler Implementation

18. for c in activeColumns(t)
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.             if s.sequenceSegment == true then
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.     if buPredicted == false then
31.         for i = 0 to cellsPerColumn - 1
32.             activeState(c, i, t) = 1
33.     if lcChosen == false then
34.         l,s = getBestMatchingCell(c, t-1)
35.         learnState(c, i, t) = 1
36.         sUpdate = getSegmentActiveSynapses {c, i, s, t-1, true}
37.         sUpdate.sequenceSegment = true
38.         segmentUpdateList.add(sUpdate)

c: a column index, i: a cell index
1) If the bottom-up input was predicted by any cell, then those cells become active

2) If the bottom-up input was unexpected, then each cell in the column becomes active
From 1), if active cell’s segment became active from cells chosen with learnState on, this cell is selected as the learning cell.

- predictive cell
- active cell with learning state 0
- active cell with learning state 1
- inactive segment
- active segment
- inactive synapse
- active synapse
From 1), if active cell’s segment became active from cells chosen with learnState on, this cell is selected as the learning cell.

- Predictive cell
- Active cell with learnState 0
- Active cell with learnState 1
- Inactive segment
- Active segment
- Inactive synapse
- Active synapse
From 2) and 3), the best matching cell is chosen as the learning cell. The cell with the best matching segment that is the one with the largest number of active synapses.
From 2) and 3), the best matching cell is chosen as the learning cell. The cell with the best matching segment that is the one with the largest number of active synapses.
Phase 2: Compute the predicted state, $\text{predictiveState}(t)$, for each cell

- This phase calculates the predictive state for each cell.
- A cell will turn on its predictiveState output if any one of its segments becomes active, i.e. if enough of its lateral inputs are currently active due to FF input.
The **predictiveState** is on; the cell queues up the following changes:

2) Reinforcement of the currently active segment, and
3) Reinforcement of a segment that could have predicted this activation, i.e. a segment that has a match to activity during the previous time step.

---

**Temporal Pooler Implementation**

---

```python
42. for c, i in cells
43.     for s in segments(c, i)
44.         if segmentActive(s, t, activeState) then
45.             predictiveState(c, i, t) = 1
46.         activeUpdate = getSegmentActiveSynapses(c, i, s, t, false)
47.         segmentUpdateList.add(activeUpdate)
48.     predSegment = getBestMatchingSegment(c, i, t-1)
49.     predUpdate = getSegmentActiveSynapses(c, i, predSegment, t-1, true)
50.     segmentUpdateList.add(predUpdate)
```

---

1) **Calculate predictive state**

2) **Update segment list**

---

![Diagram showing the implementation process](diagram.png)
1) A cell will turn on its predictive state output if one of its segments become active (OR operation)
1) A cell will turn on its predictive state output if one of its segments become active (OR operation)
2) Reinforcement of the currently active segment (at $t$)

3) Reinforcement of a segment that could have predicted this activation (at $t-1$)
Phase 3: Update synapses

- This phase actually carries out learning:
  1) Segment updates that have been queued up are actually implemented once we get FF input and the cell is chosen as a learning cell
  2) Otherwise, if the cell ever stops predicting for any reason, we negatively reinforce the segments
1) Synapses on the active list get their permanence counts incremented
2) Otherwise, negatively reinforce the segments that could have predicted this activation (at $t-1$)
Conclusion

- HTM is a machine learning technology that aims to capture the structural and algorithmic properties of the neocortex.

- HTM learns spatial-temporal patterns using two kinds of inner algorithms called respectively Spatial Pooling and Temporal Pooling.

- Through SP, HTM represents an input into a SDR.

- Through TP, HTM infer the context of current input, by activating cells in active columns.

- Through TP, HTM predicts the next-coming input, by changing cell states into predictive states.