EPL using Actors (A-EPL)

June 29, 2020

Srinivas Vamsi Parasa (parasa.vamsi@gmail.com)

Problem Statement: Given an odor which is represented by its corresponding sensor array readings (usually an array of N integers), how to memorize/learn the odor using few "sniffs" and recall the stored odor when an occluded (i.e. noise corrupted) version of the same odor is presented?

Solution: Below an algorithm is presented which is inspired by how the spiking version of the EPL works. Note that, this version doesn't use spiking neurons. Instead, it uses the actor model of computation [1].

1. Building intuition: N=2 sensors and only one noisy sensor

Let's consider a simple odor which is represented by just N=2 sensor readings, say odor1 = [4, 7] and odor2 = [9, 2], etc. Let's consider the case where only the first sensor reading is occluded (i.e. add impulse noise) on purpose for the sake of testing while the second sensor reading is left alone – i.e. always reliable. For example, odor1 = [**4**, **7**] could be corrupted by noise and we can generate sample test odors like [**3**, **7**] or [**9**, **7**], etc. Generally each sensor reading is an integer s, where $0 \leq s \leq MAX\_VALUE$.

Question: How then do we learn a pure odor and recall the same when a noise corrupted test sample of the pure odor is given?

Solution: Since the second sensor reading is always stable and reliable, we can store a mapping between the value of 2nd sensor reading and the 1st sensor reading. This will be implemented using an MC-GC network to be described now.

1.1 MC-GC network: For odors represented by N sensor readings, let's consider N Mitral Cells (MCs). MCs can be considered as the input layer for this network. Shown below in figure (1) is an MC actor network for N=2 sensor array being presented an odor o1 = [4, 7].

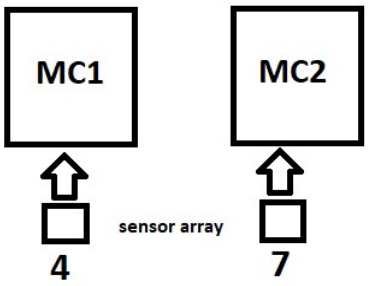

Figure 1: MC network receiving sensor array input

Each MC actor has a corresponding GC_BLOB actor associated with it as shown in figure (2). GC stands for Granule Cells. Generally, each MC is surrounded by a blob of GCs. Each GC_BLOB receives messages from

both its local MC and remote or distant MCs as shown in figure (2) below. The following are the type of connections:

(1) The messaging between a GC_BLOB and its local MC (i.e. say between GC_BLOB1 and MC1) are bidirectional.
(2) The messaging between a GC_BLOB and its remote MCs (i.e. say between GC_BLOB1 and MC2) are unidirectional - only the remote MCs send messages to distant GC_BLOBs.

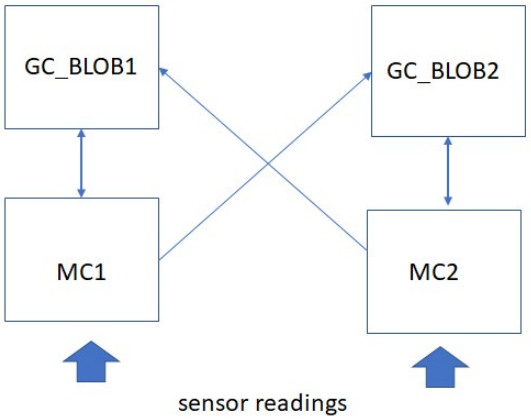

Figure 2: MC-GC actor network

1.2 Learning/Memorization:

The learning mechanism will be described now. It's best to call it "memorization" for reasons to be explained soon. When the sensor readings are presented to the network as shown in figure (2), each GC_BLOB receives messages from the MCs actors. Let's learn/memorize just one new odor: odor1 = [4, 7] as shown in figure (3).

MC actor logic: Each MC actor will simply store a local copy of the sensor reading and forward the value of the sensor reading it receives to its local GC_BLOB and remote GC_BLOBs.

GC_BLOB actor logic: Each GC_BLOB will create a dictionary (also called hashmap or key-value store) and save it in memory as shown in figure (3). Thus, GC_BLOB1 stores a mapping which says: IF my remote MC value is MC2=7, then my local MC value for this odor SHOULD BE MC1=4. Likewise, GC_BLOB2 notes that, IF my remote MC value is MC1=4, then my local MC value for this odor SHOULD BE MC2=7. The mapping is always between remote_mc_values <-> local_mc_values. Learning here is keeping track of the 'correlations' amongst various sensor values. Thus, even if the local MC value is corrupted, the corresponding GC_BLOB can use the dictionary to correct the local MC sensor reading.

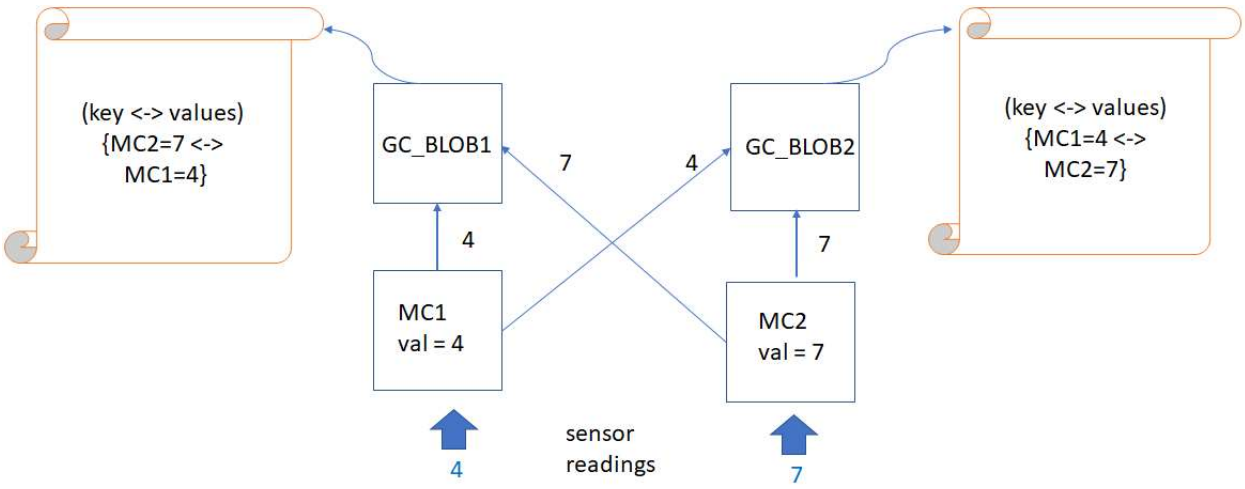

Figure 3: The key <-> value mapping stored in each GC_BLOB

That's it – the 'learning' is complete. Since this involves 'storing' the key-value mapping, it can also be called as memorization.

1.3 Why no STDP?

The learning mechanism described so far is quite simple – simple memory operations. Arrays, Linked lists and dictionaries, etc. are simple data structures for Von-Neuman computers. However, it's not so straight forward for (spiking) neural networks because symbol processing is not the forte of (current) neural computing algorithms. It can be argued that the high-level goal of STDP in the original spiking EPL learning is to indirectly create the dictionary entries which were described before. In fact, the dictionary based 'learning' is completely inspired in the process of trying to understand what the STDP learning is trying to achieve in the original spiking EPL algorithm.

1.4 Inference/Recall:

Let's present an occluded odor and see how we can do recall. For example, consider that the learned odor = [4, 7]. Since the 1st sensor is noisy as mentioned before, let it be occluded to create a new test odor = [9, 7]. In figure (4), we see how the messages flow. Each GC_BLOB will read the sensor values from its local and remote values and lookup the stored mapping/dictionary.

Cycle 1:

Each MC simply makes a local copy of the sensor value and sends its sensor value as a message to its local and remote GC_BLOBs.

GC_BLOB2 notes that it doesn't have a key stored in the dictionary as MC1=9. It has only one key stored (MC1=4). Thus, since there's a key mismatch, GC_BLOB2 will remain silent and do nothing.

GC_BLOB1 notes that it has a key stored in the dictionary for MC2=7 but notes that the value in the dictionary doesn't match. The expected value is MC1=4 BUT it receives MC1=9 instead. Since there's a value mismatch, GC_BLOB1 sends a message to its local MC i.e. MC1 to 'correct' its sensor reading to the value stored in the dictionary. After this, the state of MC1 will be MC1=4 as shown in figure (5).

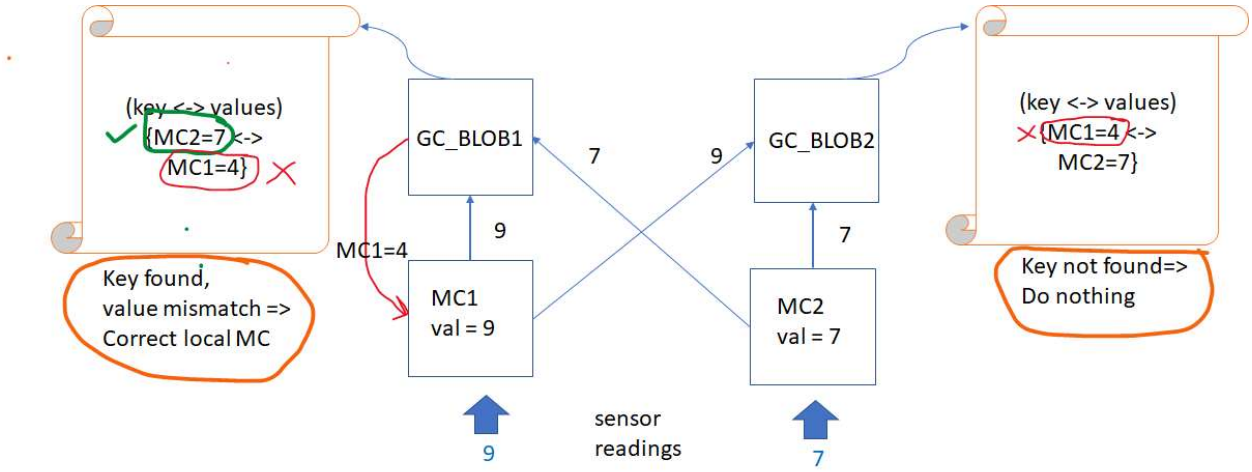

Figure 4: cycle 1 of the inference process

Cycle 2:

Now again, we repeat the steps in cycle 1. Now that the value of MC1 has been corrected from MC=9 to MC=4, we once again transmit messages from MCs to the GC_BLOBs. Now we observe that from figure (5), both the key and value matches in both the GC_BLOBs and thus we have successfully cleaned up the odor. So we've recalled the stored odor i.e. odor1 = [4,7]. This is quite trivial if only one odor is stored.

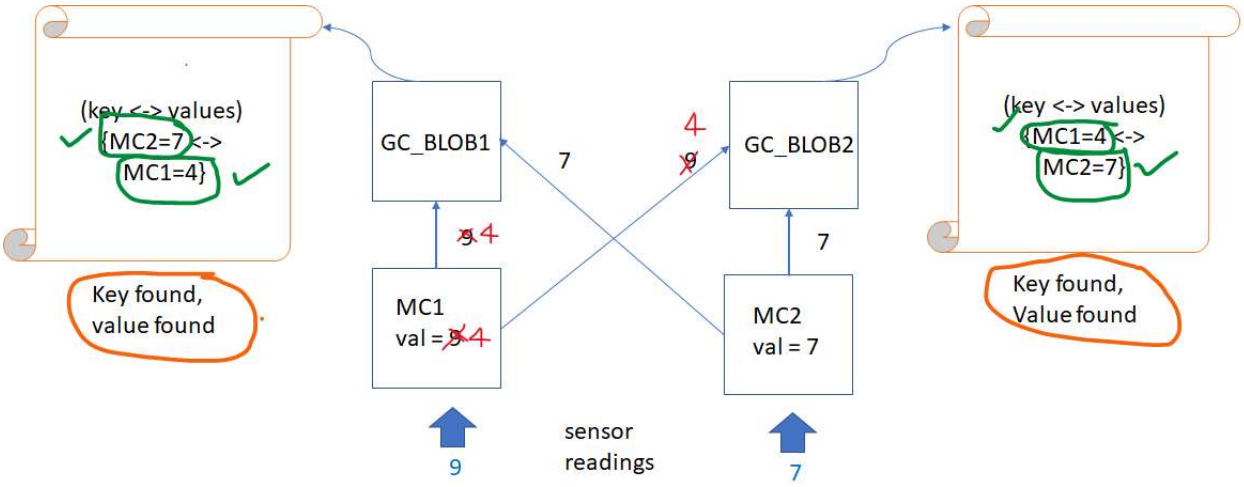

Figure 5: cycle 2 of the inference. The odor is now cleaned up

Summary of steps in inference:

(1) In each cycle, each MC will update its local copy of the sensor reading and forward its MC value to its local and remote GC_BLOBs

(2) Each GC_BLOB will check its dictionary:

i) If key not found => do nothing

ii) If key found => read the value. If value matches => do nothing; else update the local MC with the value stored in the dictionary (during training)

(3) Repeat steps 1-3 until the state of ALL GC_BLOBs is: key_found = true and value_found = true.

(4) If the above steps 1-3 are running in an infinite loop (due to reasons to be explained later), then we terminate after some constant=MAX_ITERATIONS.

2. Multiple Odors and Neurogenesis

2.1 Learning: We now extend the network to handle multiple odors. Learning a new odor requires a new entry in the mapping/dictionary because we use the dictionary to correct the noisy sensor readings. For N=2 sensors and p=2 odors, we show the network below in figure (6). Note that, we first train the network as before for odor1 = [4, 7]. After that, for training the new odor2 = [8, 1], we simply create a new entry in the GC_BLOB of each MC as shown in figure (6). In the original spiking EPL, this is achieved via neurogenesis - where new GC blobs are created for the purpose of storing the mappings/dictionary.

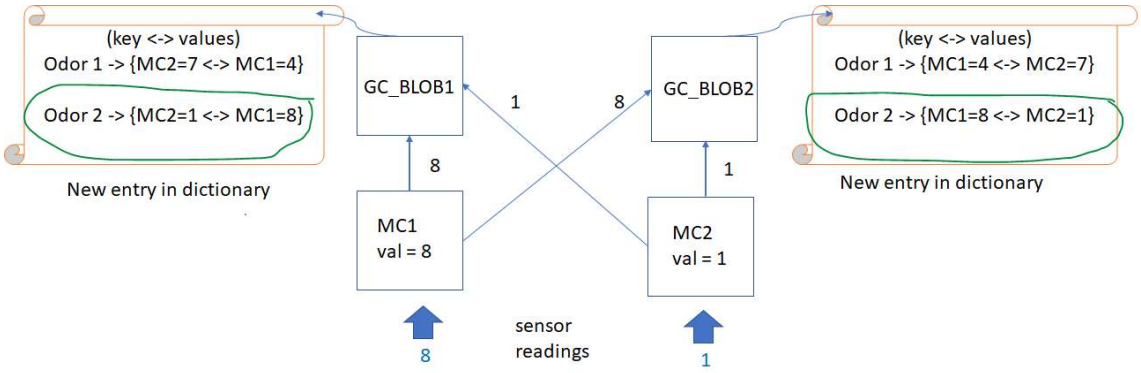

Figure 6: learning new odors – adding a new entry in the dictionary

2.2 Inference: Once we trained the network to learn multiple odors (N=2 sensors and p=2 odors), we can now present the network with noise corrupted versions of the two learned odors [4,7] and [8,1] and see how it works.

We can describe the network dynamics under various possible scenarios:

1) **No noise:** If we present one of the trained samples as a test sample to the network, then the network activity will stop after 1 cycle as both the GC_BLOBs will be in the key_found=true and value_found=true state.
2) **Only one noisy sensor:** if only one of the sensor readings is corrupted, then the network activity will converge after 2 cycles after the faulty sensor reading is corrected. For example, if the trained odors are [4,7] and [8,1], then if we present [8, 5] as the test odor, then we show the cycle 1 of inference in figure (7) and cycle 2 of inference in figure (8). After cycle 1, the second sensor reading which is faulty is corrected and in cycle 2, we see that the odor is cleaned up and matches with odor2 ([8,1])
3) **All (N=2) noisy sensors:** If both the sensor readings are noisy and none of them match with the keys of both the GC_BLOBs then the network will come to a halt after 1 cycle. In section 3, we'll describe a technique to reduce the probability that ALL of the incoming MC sensor readings are corrupted/noisy to a very small value.

4) **Special case – infinite loop:** If the test odor is a mix of both the trained odors, then the network will continue to oscillate in an infinite loop. For example, if the trained odors are [4,7] and [8,1]. If we present [4, 1] or [8, 7] as the test odor, then after the 1st cycle when the values of the MCs are corrected they oscillate between [8, 7] and [4, 1] due to repeated correction of the MC values in each cycle. To avoid such infinite loops, we terminate the inference after MAX_ITERATIONS (usually 5 cycles is enough). Also, there are other techniques to avoid/mitigate this scenario of unstable oscillations and will be described later.

Readout: To make it easier for the readout i.e. to which trained odor the test odor matches with, we can have an additional variable called odor_id in each GC_BLOB. After every cycle, we assign the odor_id to the odor_id for which the dictionary keys and values match. After MAX_ITERATIONS we can read the value of odor_od from each GC_BLOB and use a majority vote to decide the final odor_id. This will be explained later for odors with large number of sensor readings.

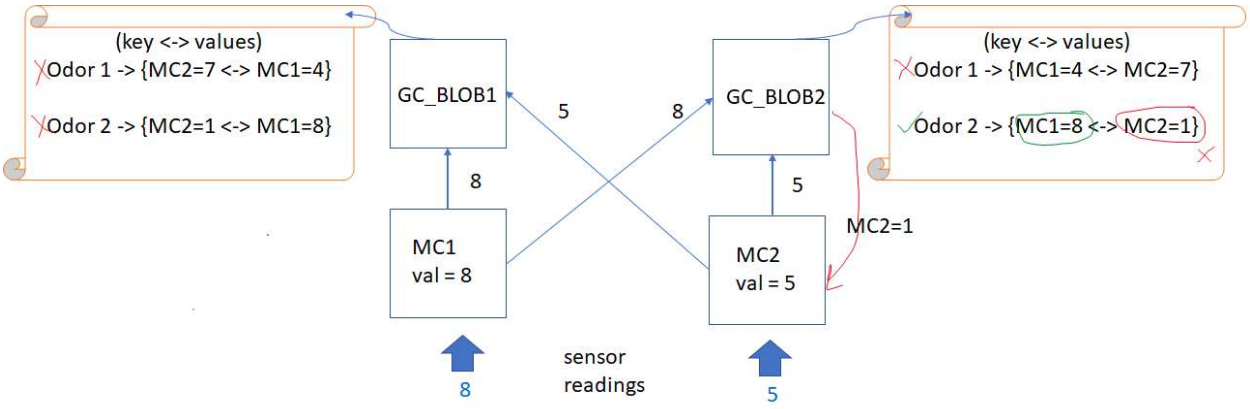

Fig 7: cycle 1 of inference

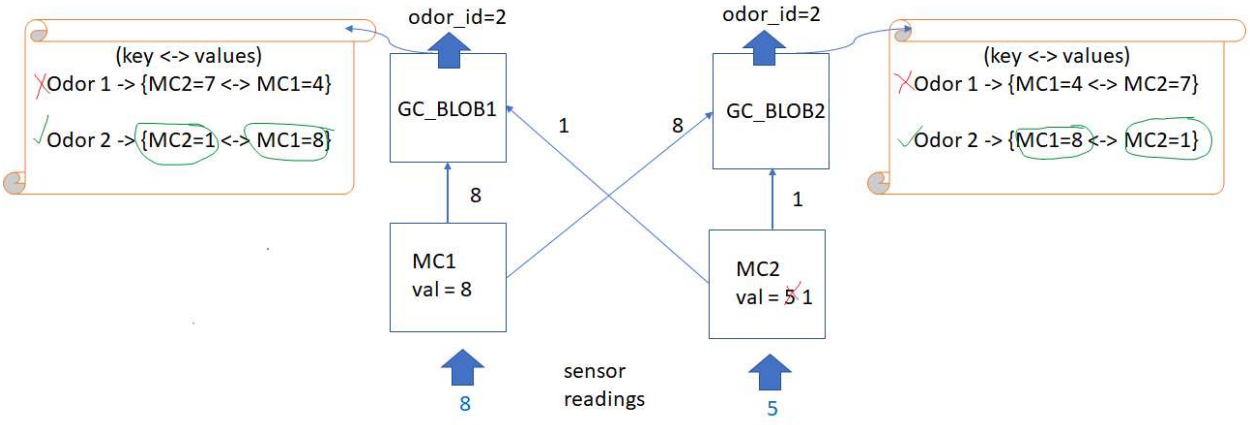

Figure 8: Cycle 2 of inference

3. EPL for large sensor arrays

Having developed an intuitive understanding of this algorithm of N=2 sensors, we now describe the additional features of the algorithm needed to make it work for large sensor arrays. For example, the human nose has about N=400 nasal receptors which can distinguish between 1 trillion odors[2]. In EPL, we use a gas sensor array of size N=72 with 50% sparsity for the sensor readings. In figure (9) we show an example network with N=100 sensors.

3.1 Receptive field of each GC_BLOB

If we want to train the network to learn p odors for size=N sensor readings, then if each GC_BLOB dictionary entry keeps track of all the remaining N-1 sensor readings, then to learn each odor will require $O(N^2)$ memory. For a total of p odors, it'll be $O(pN^2)$. We can not only reduce this memory requirement but also improve the speed of execution (faster dictionary lookup) by using a smaller receptive field of MC actors feeding into each GC_BLOB actor. Thus, instead of all the N sensor readings being sent to each GC_BLOB, we only connect a small fraction say n=fN (0<f <=1) MCs to any particular GC_BLOB. We show an n=5 receptive field in figure (9). We can use random connectivity for this. Thus, for each of the p odors, a randomly generated receptive field of n(=fN) remote MCs will be connected to each GC_BLOB.

What is a good value for f – the connection probability?

If ALL the sensor readings of the receptive field of a GC_BLOB are corrupted then the GC_BLOB can no longer correct its local MC value, which in turn is forwarded to other remote GC_BLOBs. Thus, this impairs the odor cleanup process during inference.

If each GC_BLOB receives n=fN connections from other MCs, then in the absence of additional information (say, prior occlusion probability of each sensor), the probability that ALL of the n sensors would be corrupted is given by an exponentially decreasing function:

$$prob = \frac{\binom{n}{0}}{\binom{n}{0} + \binom{n}{1} + \cdots + \binom{n}{n-1} + \binom{n}{n}} = \frac{1}{2^n}$$

If N=72 and f = 0.3, then n ≅ 21. The prob is $\frac{1}{2^{21}} = 4.7 \times 10^{-7}$. Thus, along with 70% savings in memory, we also get a very low probability that ALL the sensor readings will be corrupted. Note that in the presence of high occlusion noise (say 90%) and high sensor sparsity (50%), we need larger receptive fields for efficiently cleaning up the odors. Thus n=fN is a parameter that needs to be tuned.

3.2 Example of a GC_BLOB with large receptive field

In figure (9), we see a snapshot of a network with N=100, p=20 odors and n = 5 during inference phase. It has already learned 20 odors. Each GC_BLOB has a separate receptive field (of n=5 MCs) for each odor stored.

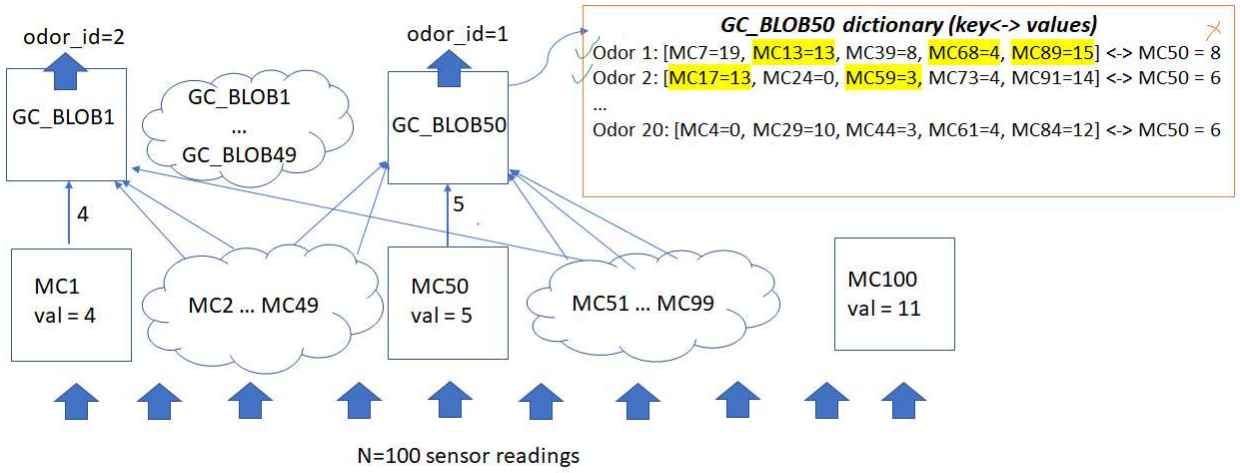

Figure 9: Snapshot of an example network for N=100, p=20 odors and n = 5 (f=0.05) during inference phase

3.2 Breaking ties for multiple odors matching dictionary entries

In the example network described in figure (9), it is quite possible that there could be multiple matching dictionary entries (shown in yellow in figure (9)). Based on the dictionary for GC_BLOB50 in figure (9), should we assign GC_BLOB50 to odor1 or odor2?

For odor 1, we see that there are 3 key matches out of n=5 MC sensor receptive field (i.e. two sensor readings got corrupted) and for odor2 we see that there are only 2 key matches (i.e. three sensor readings got corrupted) … in the absence of additional information, we'll simply break the tie with a majority vote and thus, in this cycle of inference, we assign GC_BLOB50 to odor1 as shown in figure (9). Now, since there's a value mismatch for odor1 (although it has the winning set of keys), we correct the value of the local MC: MC50=5 to MC50=8 as shown in figure (9). Now after MC50 is cleaned up, it could potentially trigger key matches in the dictionaries of the GC_BLOBs to which it feeds to. Thus, this process of cleanup will continue for multiple cycles. Hopefully, at the end of many cycles, we can clean up the odor and all or almost all of the GC_BLOBs point to the same odor_id. Usually, in practice, MAX_ITERATIONS=5 is sufficient for a large majority of the GC_BLOBS (>95%) to point to the same odor_id.

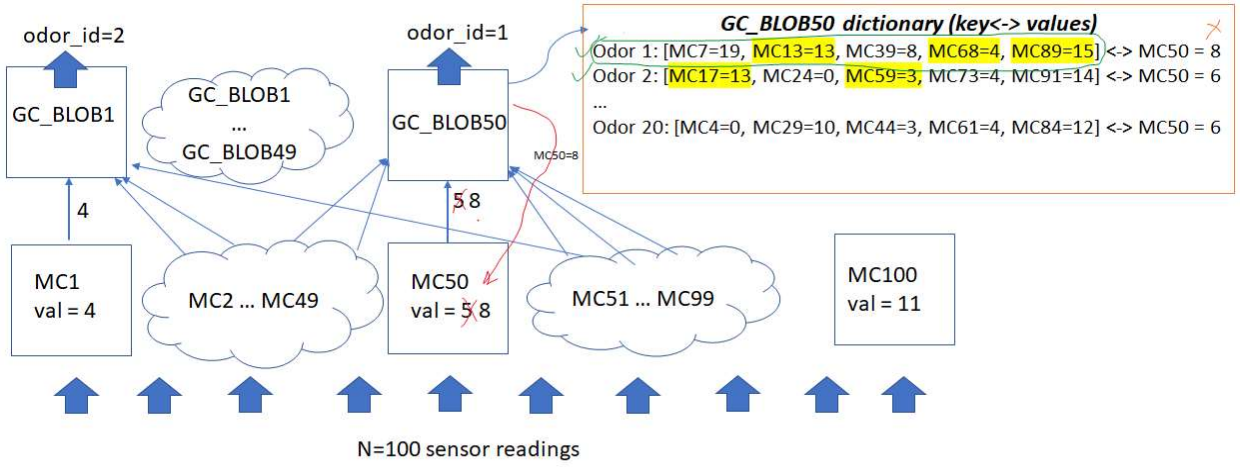

Figure 9: Network show in in figure (8). How the local MC value is corrected after breaking the tie between dictionary matches

4. Summary of EPL using actors

Strictly speaking there is no need to have a separate MC and its corresponding GC_BLOB actor. We only used it because the original spiking EPL algorithm uses them. For our purposes we can combine MC and its GC_BLOB into one actor. Nevertheless, we'll still use separate MC and GC_BLOB actors in further discussion.

We need the following actors:

1) **MC actor**

   Learning: receives sensor readings, makes a local copy and forwards the value to its local GC_BLOB and remote GC_BLOBs. Each GC_BLOB receives messages from n=fN remote MCs (receptive field).

   Inference: receives sensor readings, makes a local copy and forwards the value to its local GC_BLOB and remote GC_BLOBs. Also, if its local GC_BLOB tells it to correct its local sensor value, it will update its local copy before forwarding to remote GC_BLOBs.

2) **GC_BLOB actor**

   Learning: each GC_BLOB receives sensor readings from its receptive field of n remote MCs. Creates a dictionary whose keys are id and sensor readings from remote MCs. The value is the sensor reading from the local MC. For each odor, create a new dictionary entry. Each entry is a collection of key <-> value pairs.

   Inference: each GC_BLOB receives sensor readings from its receptive field of n remote MCs and also its local MC. Pick the dictionary entry which has the maximum number of key set matches.

Look up the dictionary value associated with the winning set of keys. Then if the value stored is different from the copy stored in the local MC, then tell the local MC to update its local copy before forwarding to other GC_BLOBs.

3) **Network actor (optional)**
   This actor helps to 'route' the messages between MCs and GC_BLOBs. It's basically the network topology. In the neuromorphic chip, this is taken care of by the hardware (includes on-chip mesh).

4) **Executor actor (optional)**
   This actor can be the user facing actor which receives commands about the mode of operation – learning or inference. It can also synchronize the operation of all the other actors. It sends commands to the MC and GC_BLOB actors. It can receive sensor readings from the user/ sensor hardware and forward them to the MC actors, etc. It can also do the readout after inference. In the neuromorphic chip, this is optional.

5. Conclusion

The basic working of the EPL algorithm using actors is described. More details will be provided in future documentation or presentations.

References

[1] Actor Model of Computation, https://en.wikipedia.org/wiki/Actor_model

[2] Human Nose can detect 1 trillion smells, https://www.sciencemag.org/news/2014/03/human-nose-can-detect-trillion-smells