Cblock with buffer – summary of changes

Martin Takac, 19/03/19

1 Introduction

This document is for people already familiar with the previous version of cblock. It doesn't repeat principles explained in the previous documentation, but it rather focuses on new features. If you are not familiar with cblock, please refer to the following resources (on Cogarch team google drive) first:

- Doc/cblock-chunking/cblock.pdf – self-contained manual to the first cblock version (without buffer). Most of the stuff therein is still valid, new stuff is described in this document.
- Videos/cblock1_basics.mp4 – a video introducing cblock basic concepts – next element prediction, sequencing and chunking, surprise, entropy and chunk boundaries. It demonstrates the concepts on letter chunking example. This is the one you should watch first.
- Videos/cblock2_neuroeconomics.mp4 – a video introducing more advanced cblock concepts – difference-based planning, actual state vs. desired goal state, plan effects, goal-driven behaviour, attentional weights on desired change along different dimensions of the state space (neuroeconomics). It demonstrates the concepts on drawing chunking example. Watch this one as a sequel to the above one.

There are also two videos for the new cblock with buffer:

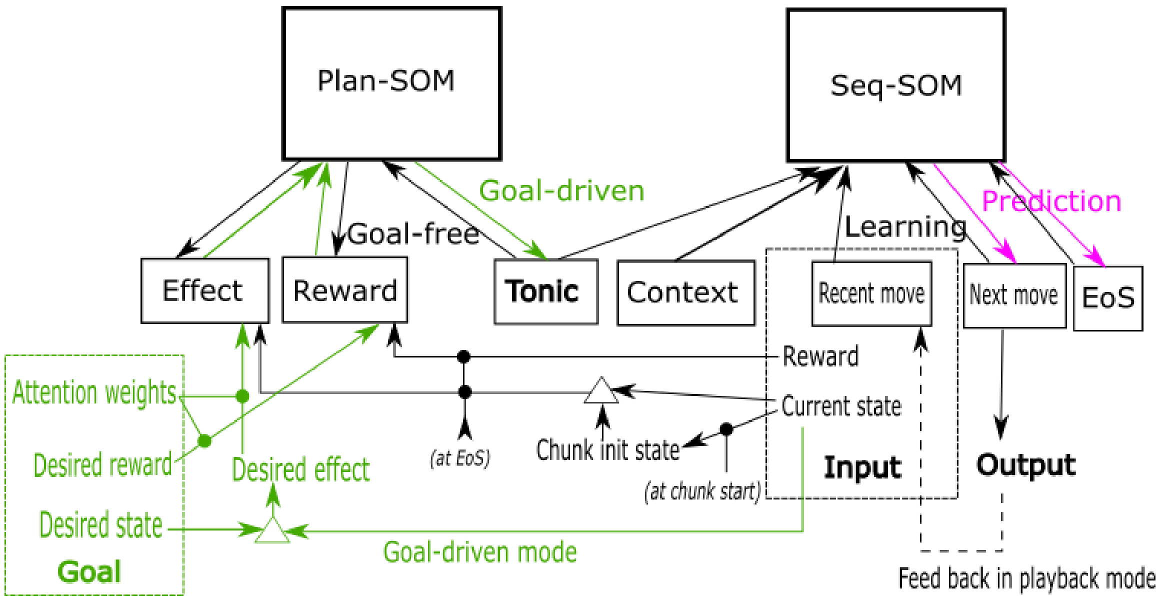
- Videos/cblock_with_buffer/cblock_buf_basics – demonstration of new features of cblock with buffer on the example of letter sequences.
- Videos/cblock_with_buffer/cblock_buf_technical – very important video describing file structure of cblock, its input-output interface and how to connect it.

These videos are reasonably self-contained, here I summarize main points in writing.

2 Main changes

Buffer for sequences added

Cblock still consists of two asoms: sequencing SOM (seq-SOM) and planning SOM (plan-SOM), see the picture below. The former learns transitions between sequence elements and the latter declarative representation of whole sequences (plans) associated with effects and rewards. Previously sequencing SOM was learning all the time, i.e. it stored all *observed* transitions between sequence elements. This is could get cluttered by rubbish or could predict sequences that lead to undesirable effects. Now it is different. A sequence is first stored in a buffer and the user has a full control of when it should end, and whether the buffered sequence should be learned at that point or discarded. In this way the sequencing SOM will only learn *good* sequences, i.e. those deemed good by the user. The cblock doesn't decide autonomously any more: it will still signal arrival of a surprising element or expectation of a sequence boundary, but decision is now external to cblock.



Prediction is separate from training

An unexpected benefit of the buffer is that the prediction is now completely separate from storing the actual sequence: Each new incoming element is added to the buffer and to the evolving declarative/tonic representation of the whole sequence. When the user decides to finalize the sequence, the buffer contains the sequence as it actually happened, along with its recorded declarative representation. Independently of that, with each new incoming element, cblock tries to predict the most likely next element. For this it uses both sequencing SOM and plan SOM:

In case the element that just arrived is not surprising (see below), cblock takes the evolving fragment of the tonic representation in the buffer and queries the plan-SOM for a complete plan consistent with the fragment. Then the seq-SOM is queried with the retrieved plan together with current “context” and “recent” inputs for the most likely next element (which can be a proper one, or an end-of-sequence, see below). This element is returned on the cblock output as its prediction.

In case the incoming element doesn’t match the prediction from the previous time step (based on an over-threshold multiple of sliding average of KL divergence between the predicted distribution and the actual element), the cblock signals a surprise. In this case it sets the seq-SOM’s alpha for tonic input to zero and tries to predict the most likely tonic from the recent element and its context. This (soft-output) tonic then queries plan-SOM for (hard-output) best-matching stored plan. This plan is in turn sent back to tonic input of the seq-SOM where it (together with the context and recent) retrieves a predicted next element.

Thus, because the sequence elements and the actual tonic are stored in the buffer, the seq-SOM inputs can be tweaked in any desirable way that helps the prediction in the meantime without affecting seq-SOM learning.

Inference of the most likely plan, effect, reward

Because the plan-SOM is queried for a stored plan consistent with the evolving fragment, a side effect of this is intention recognition – whether cblock is surprised or not, it returns on its output also the most likely effect and reward a retrieved plan was stored with. This is helpful in goal-driven mode: if the cblock is pursuing a plan to satisfy a goal but is surprised (e.g. because the plan is an alternating sequence of user/avatar actions as in dialog and the user does something unexpected), it tries to recover by inferring the most likely new plan and reacts consistently with that. At the same time it signals surprise and returns the most likely effect and reward, so that the user can decide whether to insist on following the original goal or go along with the new plan – here the control is on the user too: it is up to her to set the planning/goal input in the next step (and discard or finalize the sequence).

Note: because internally the effect=final_state–initial_state and desired_effect=goal_state-current_state, in case you want to install the inferred effect as the new plan, set goal_state to inferred_effect+current_state.

More consistent tonic input in the seq-SOM

In the old cblock, the seq-SOM was trained online as the elements were arriving. That meant that the tonic representation of the whole sequence had to be used before it was complete: each transition of a sequence only contained the fragment of the whole sequence seen so far in its tonic. That caused instability in retrieval (goal-driven replay), because then the plan was complete, so the tonic input only partially matched the stored one. With the buffer, the seq-SOM is only trained after the whole sequence has been seen (i.e. when the user decides it is complete and sends the command to finalize it), so the training tonic input is the same for all transitions and equal to the one that will be used during replay – the complete declarative representation. This gives much fewer retrieval errors.

Result, reward and eos needn't be in sync with the last element of the sequence

From the point of view of interfacing with cblock, now the demand to end the sequence (either by discarding the buffer content or using it to train the SOMs) is a control command separate from incoming elements. In the old cblock it had to come in sync with the arrival of the last element of the sequence, which was problematic. At the end of the sequence cblock looks at the values of reward and state and stores them with the plan in the plan-SOM (state is stored as effect=state-initial_state, where initial_state was remembered after the last sequence end). In reality, the reward and change of state happen some time after the last action in the sequence, not in sync with it. Because now the sequence will only end when the user says so, the reward and state change can arrive *after* the last element (actually, the decision to end the sequence can be based on them, e.g. on arrival of a particularly high reward).

Explicit prediction of eos, separate from elements

Because end of the sequence now happens after the last element, it can be stored as a separate transition in the seq-SOM. Hence, the seq-SOM will in turn predict all the elements in the sequence and then the EoS after the last element, e.g. J -> O -> H -> N -> EoS. Again, the responsibility for what to do when the EoS is predicted lies outside cblock: it can be either ignored or fed back to cblock inputs (during playback), either in the form of discard, or learn command.

3 How to connect the cblock

Same as in the old version, the whole system sits in cblock folder. It communicates with the rest of the system via cblockInputs.blm and cblockOutputs.blm sitting in the root of cblock. Please watch Videos/cblock_with_buffer/cblock_buf_technical on Cogarch team google drive for detailed explanation of new interface. Useful information can also be found in Doc/cblock-chunking/cblock.pdf, modulo the changed described here and in the video.

Note: If you find out that the system works better for other than default values of some parameters, please do not change them directly in cblock folder, but via a config_cblock.blc connector from above to avoid that your values will be overwritten next time I update cblock.

cblockInputs

In contrast to previous version, cblock can take three kinds of input, whenever cblockInputs/ready is set high (the synchronization mechanism via ready-reset still works as in the previous version):

inputType_nextElem – a new element arrived,

inputType_resetSeq – a control signal saying that the buffer content should be discarded without training seq-SOM

inputType_finalizeSeq – a control signal saying the sequence in the buffer was successful and should be stored in seq-SOM (and its plan, effect and reward in plan-SOM)

It is the responsibility of the user to make sure exactly one of these three variables is 1.

State and reward can be connected all the time, its change doesn't require raising the ready signal, but cblock will only attend to them when needed:

- When finalizing the sequence to compute its effect and store it along with the reward and the plan in the plan-SOM
- When starting a new sequence to remember its initial state,
- With arrival of a new element in goal-driven mode to check whether the goal has been reached.

Other cblockInputs variables work as before, see the comments in the cblockInputs.blm file.

cblockOutputs

The key principle is that regardless of which of the three types of input arrived into cblock, cblock should always signal when it finished processing them with cblockOutputs/ready (again ready/reset sync mechanism works as before). In case of resetting sequence, there's no noew prediction, the cblock just acknowledges the discard has been completed. If the input was finalizeSeq, it depends on whether being in goal-driven mode. If not, there's no meaningful prediction from eos signal, so the ready just means acknowledging the sequence earning has completed. On the other hand, in goal-driven mode, each time a sequence is finalized, cblock refreshes its goal buffer and recomputes a new plan. A new plan leads to a prediction of its first element, so in this case there is a valid prediction. And there's always a valid prediction in nextElem case. Whether the cblockOutputs contains a valid prediction is signalled by cblockOutputs/contain_prediction. If it is 0, the predicted element should be ignored.

In case there is a valid prediction, it is either prediction of a proper element, or of end-of-sequence. This is signalled by cblockOutputs/eos_predicted. In case it is high, the predicted element (bitmap or x,y) should be ignored.

With prediction, values of good_enough, plan_good_enough and goal_reached are returned. In case of playback, the predicted element should only be executed and send back to input if good_enough (i.e. low entropy), plan_good_enough (in non goal-driven mode always, in goal-driven based on over-threshold value of winner_activity_raw in the plan-SOM, i.e. whether the retrieved plan satisfies the requirements well) and not goal-reached (do nothing, if the difference between desired state and the current state is below threshold). However, this happens outside cblock, so again it is up to the user how (s)he wants to use these values or whether to ignore them.

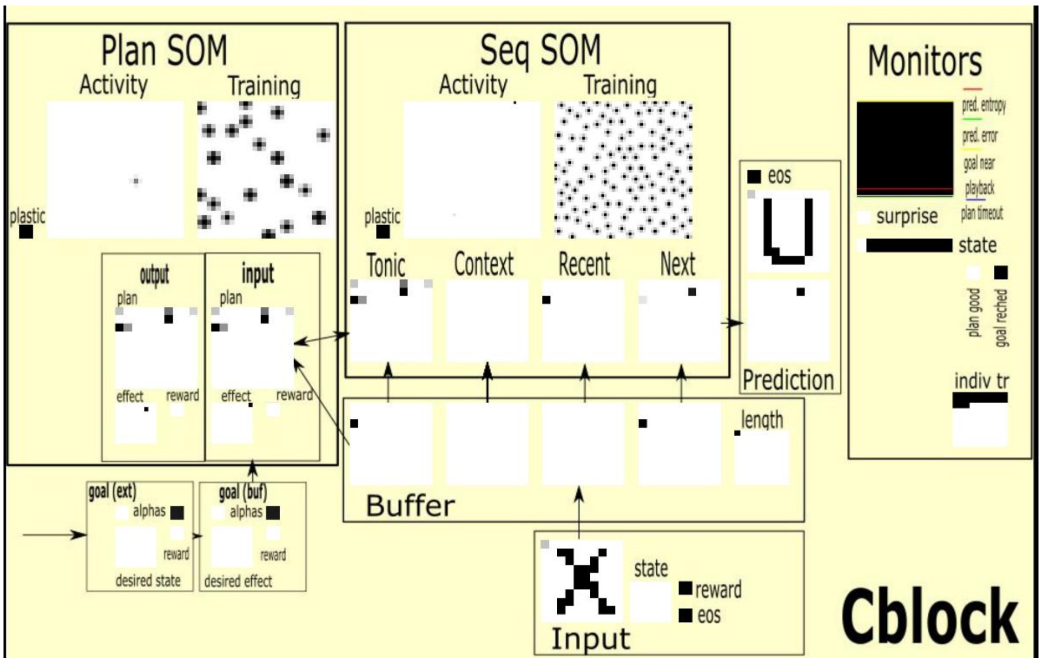
There are two alert variables signalling that the size of the buffer or size of the individuation SOM are not enough for this application – see the comments in cblockOutputs.blm for details.

Cblock also signals when the element was surprising (surprise) and what plan it is most likely part of (inferred_plan_effect and inferred_plan_reward, even in case it was not surprising).

4 Displays vs debug displays

All displays are now separate from cblock and sit in the same directory as cblock. There are two options:

1) A new futuristic display created by Oleg Efimov. In case you want to use it, take any of my new cblock demos (letters or drawing), and copy (along cblock folder) also the folder displays, and the files displayConstants.blm, display_cblock.blc, visibility.blc, and change the default value of displayConstants/is_visible to 1. The qml control for these displays is in Display.qml.
2) Debugging displays created by me (see Figure below). Not that fancy, but more detailed. If you want to use these, copy (along cblock) the folder debug_displays and the connector debug_display.blc


Plan SOM
Activity
Training
plastic
output
plan
effect
reward
input
plan
effect
reward
goal (ext)
alphas
reward
desired state
goal (buf)
alphas
reward
desired effect
Seq SOM
Activity
Training
plastic
Tonic
Context
Recent
Next
eos
Prediction
Buffer
length
state
reward
eos
Input
Monitors
pred. entropy
pred. error
goal near
playback
plan timeout
surprise
state
plan good
goal reched
indiv tr
Cblock