Cblock for chunking, planning and goal-driven behaviour

Martin Takac, 31/01/19

1 Introduction

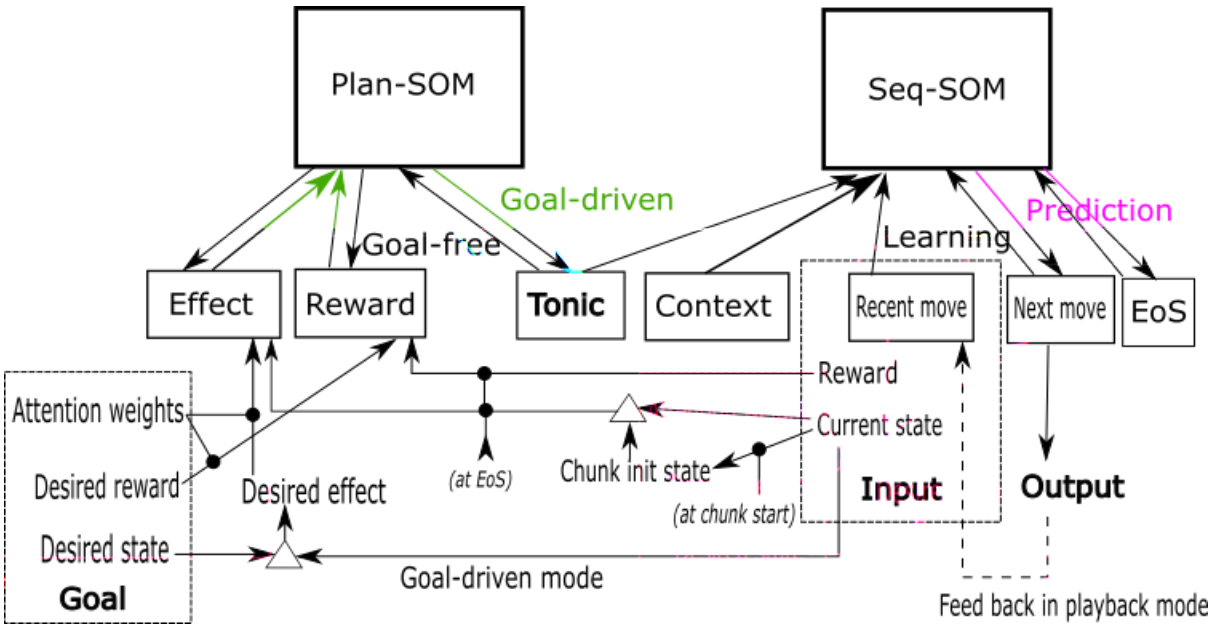
Cblock is a model fragment / building block combining several powerful and cognitively relevant features. It can:

- **learn sequential dependences** in incoming data and **predict** probability distribution over possible **next inputs**,
- notice repeatedly occurring sequences, automatically **detect sequence boundaries** (based on surprise in prediction) and represent sequences as **chunks/plans** for future execution/replay,
- associate plans with reward and their effects on the system state,
- **recognize** possible **plan**, **intent** (effect) and expected **reward** from a fragment of ongoing sequence,
- implement **goal-driven behaviour**: find and execute a plan that can reduce most differences between a current system state and a desired state (goal). Differences are weighted by individual attentional weights.

It resides in a self-contained folder with well-defined I/O interface, so can be replicated as many times as necessary. The interface is described later in this document.

2 Main features

The architecture of the cblock is on the following picture:



It consists of two asoms: sequencing SOM (seq-SOM) and planning SOM (plan-SOM). The seq-SOM learns to predict the next element in a sequence and detects the sequence boundary when a newly arrived element is surprising (doesn't match prediction) or when an external criterion (e.g. an unusually high reward, or anything else the user connects to the end-of-sequence (EoS) signal) is met. Each new element comes with a 100-dimensional system state and a (scalar) numerical reward.

At the end of a sequence, a declarative representation of the whole sequence is stored as a chunk in the plan-SOM, together with its effect (the difference between the system states at the end and at the start of the chunk) and reward at the sequence end (interim states and rewards inside the sequence are ignored). This can later serve for executing a plan with a particular effect or reward in mind (in goal-driven mode) or for recognizing what plan the developing observed sequence can be a part of (and predicting the associated reward and effect). Note that the system distinguishes between the actual reward received at each step of the sequence and anticipated reward (for completing the whole plan).

The system supports **observation** and **generation/playback mode**, but this distinction is external; the only difference is in how the predicted next element/move on the cblock's output used. The seq-SOM is receiving a sequence of actual elements/moves (together with system state and reward) and at each moment it predicts a likely next move which is then sent to the cblock output. The cblock is usually a part of a larger system (similarly to WM on top of CDZ in Baby-X), in which the prediction can serve as a weak (or no) top-down bias on the connected sensorimotor/behaviour-driving system (to create expectations in observation mode), or as a strong top-down bias (to generate a behaviour in generation mode). Thus, the feedback loop sending the predicted outputs back on the inputs is indirect/external, via the top-down influence. In case of a simple planning system that uses cblock, playback behaviour/mode should be realized by a connector (outside cblock) feeding the predicted output back on the input.

More important (cblock-internal) distinction is between **goal-driven** and **goal-free mode**, which determines how the **Tonic** plan medium providing input to seq-SOM is filed: In goal-free mode (black arrows in the figure), the tonic representation of the whole sequence is gradually being constructed from the incoming inputs as a decreasing gradient coded hologram and is sent as a bottom-up input to plan-SOM for plan recognition (so that the plan-SOM can predict what plan/chunk is the fragment seen so far a part of). In goal-driven mode (green arrows in the figure), the plan-som is queried with alpha-weighted desired goal and the best-matching plan is retrieved top-down and sent to seq-SOM's tonic.

2.1 Difference based plan selection

We can think about elements in the sequence as actions or moves in a game that change the state of the system, whatever the state is: drawing on a drawing board, levels of Baby's neurotransmitters or needs, or a checklist of user-filled form entries. If we represent the state of the system as a high-dimensional vector with arbitrary (application-specific) semantics, each particular state can be imagined as a point in the high-dimensional state space. A move/action potentially changes the location of that point, and a sequence/chunk creates a trajectory of points in the state space. The net effect of a plan can be computed as a difference/delta vector, subtracting the initial state from the final state of the chunk. These net effects are stored along with the "motor" plan (recipe allowing the seq-som to replay the sequence of moves) and the reward received at the end of the chunk in the plan-som. In goal-driven mode, the user provides a goal – a desired state (and/or desired reward) and

a vector of attentional weights (alphas) for each dimension of the state space, as well as mixing alphas for influence of goal-state vs. result. Desired effect is computed by finding the difference between the desired state and the current actual state. The plan-som is then queried to find the best matching plan to eliminate the current difference weighted by the attentional alphas. Once the plan is selected, it is sent as tonic input to the seq-SOM and the cblock replays it until the end of the chunk, goal state reached, plan timeout or unexpected event (whatever happens earlier). Each time a sequence is completed (whether the goal was reached or not), the system updates its representation of the goal and current state and prompts the plan-SOM to get a new plan. If the just-finished plan only eliminated some of the differences, the new plan would be selected with respect to the remaining ones. At this point, inhibition of return (IOR) applies, so that the system is less likely to select the most recent plans. This give variability in behaviour if IOR is on (cblock/planning/control/ior_decay>0).

Note: For inspection of weights of individual neurons, both seq-SOM and plan-SOM can be activated top-down via qml sliders ("inspect SOM") and a neuron position selected with a qml cursor widget. Plans for different goals and alphas can be visually inspected too, with a qml slider "plan_inspection".

3 How to connect the cblock

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. All inputs related to incoming data (the recent sequence element, system state, reward and potentially also an external signal forcing the sequence end, as well as a control signal for goal-driven vs. non-goal driven behaviour) should be written to cblockInputs.blm (see comments therein for details). Cblock is event-driven, it stays in sleeping state until cblockInputs/ready=1, then wakes up, reads in and processes the inputs. When it is done, it writes all the outputs into cblockOutputs.blm (see comments therein for details) and raises cblockOutputs/ready=1 flag. The rest of the system should read and process the outputs from there. Another entry point to the system is cblock/planning/goal.blm which specifies a desired goal (state, reward or their alpha-weighted combination, see comments in the blm file for details).

There's a bunch of parameters in cblockParams.blm that influence the behaviour of the system (specific parameters of seq-som and plan-som are in their respective subfolder/asomConsts.blm). The initial values should work reasonably well for most applications, but it can happen that for a particular type of input, some values need to be fine-tuned. In the provided examples (described later in this document), the parameters are accessible via qml interface – see create_qml.txt in the root folder of the provided examples. 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 (again, see the provided example applications for a template). In this way you'll avoid that your values will be overwritten next time I update cblock.

3.1 Three types of input conversion

For cblock to work robustly, the elements are internally represented by 1hot codes. That is why inputs that are not 1hot need to be *converted* via a SOM with a high best_matching threshold and a small sigma so that sufficiently different inputs are remembered in different neurons. The activity of the SOM then goes through wta modul to yield 1hot coded input for seq-SOM. For output conversion, there's a copy of the SOM with identical weights that is used top-down.

Cblock includes SOM conversion for two types of input: **x,y normalized positions** or **distributed 10x10 bitmaps** (like letters, images, anything). The third type is **no conversion** (just wta) for already reasonably localist 10x10 bitmaps (like activities of SOMs in use_soft_output mode=0, where the wta conversion that preserves only the winner of the activity map doesn't cause loss of data during backward output conversion). *In a dialog system, perhaps the best way is to encode dialog node IDs as 1-hot bits in the bitmap – e.g. first 50 bits for user utterances and the last 50 bits for avatar's ones: in this case a further conversion in cblock is not necessary (do_indiv=0).*

Which type of input conversion will happen, depends on static parameters cblock/cblockParams/do_indiv and cblock/cblockParams/indiv_xy (their values determine perform_time_step of the conversion SOMs):

- For X,Y input conversion, set do_indiv=1, indiv_xy=1 and connect inputs to cblockInputs/input_x and y and outputs to cblockOutputs/predicted_x and y.
- For bitmap input conversion, set do_indiv=1, indiv_xy=0 and connect inputs to cblockInputs/input_bitmap and outputs to cblockOutputs/predicted_bitmap.
- For no input conversion (just wta of bitmap), set do_indiv=0, indiv_xy=0 and connect inputs to cblockInputs/input_bitmap and outputs to cblockOutputs/predicted_bitmap.

The granularity of input conversion SOM can be tuned with cblock/IO/xy_som_consts/qmlSensitivity for xy positions or cblock/IO/bitmap_som_consts/qmlSensitivity for distributed bitmaps. Watch for how quickly the training record of the indiv-SOM (bottom right of the display) gets filled up. If too quickly, decrease the sensitivity.

3.2 More on I/O synchronization via ready and reset flags

In order to prevent unintended repeated reading of the same input,[1] the ready flag needs to be set back to zero after the input has been processed. This is signalled via a reset flag:

Whenever there's a new input element to be processed, set (from outside cblock) cblockInputs/ready=1. When cblock reads the input, it will set cblockInputs/reset=1. Listen for reset and turn ready off (again outside cblock, the best within the same container connector that set cblockInputs/ready=1), e.g. by:

```
BL_identity_connector
connect_cblock/perform_calculate=cblock/cblockInputs/reset
cblock/cblockInputs/ready=cblock/core/control/zero
```

A similar communication is for cblockOutputs, but this time it is cblock who signals ready and an external code who reads the output when ready and sends back the reset signal (do not forget to only send reset as a trigger, i.e. set it back to 0, one or two timesteps later, otherwise the cblock will never become ready again while its reset=1).[2]

Cblock always gives a prediction of what's likely next move, given the current input. If the prediction is good enough, it can drive the next input via playback. cblockOutputs/good_enough signals whether the predicted output comes from a distribution with entropy lower than cblockParams/entropy_thr. In goal-driven mode, cblockOutputs/plan_good_enough signals whether the selected plan matched

[1] Intended repeated elements are perfectly legal, such as CCGGAAG notes in Twinkle, twinkle melody.

[2] In the drawing example, the reset was causing this problem, so I do not use cblockOutputs/reset=1 at all, but it still works.

the goal sufficiently, and cblockOutputs/goal_reached whether the goal has been reached already (see comments in cblockOutputs.blm for details). The user can choose (outside cblock) to not execute/play back the predicted move if the prediction or plan are not good enough or if the goal has already been reached (for example of that, see connect_cblock.blc in any of the examples described in sections 5 and 6 – the part starting with perform_calculate=cblock/cblockOutputs/ready), otherwise the outputs is sent back to the input. The speed of playback can be regulated via a LIF: when output is ready and good, buffer it and start LIF countdown. When LIF fires, write the buffered outputs back to cblockInputs (and set its ready=1). The playback loop is external to cblock, but for an example of how to use it, see my drawing and letters examples.

3.3 Input in detail: state, reward, eos, self-gen

Seq-SOM predicts what's next all the time. When it detects an event boundary (either by surprise – mismatch in prediction, or by receiving an external end-of-sequence signal), it remembers the "motor program" in the plan-SOM, along with the plan's effect (change in state brought about by the plan and the reward received with the last element of the sequence. This will help the plan-SOM to later recognize a plan from a fragment of a motor program and predict its reward and effect (intention recognition) and also to drive behavior by goal (a particular desired state or maximum reward).

Send the appropriate reward signal (between 0,1) and system state (10x10 bitmap or any 100dim vector) along with the sequenced input when cblockInputs/ready. You can also choose to force chunk boundaries with an external signal (cblockInputs/input_is_eos=1 sent along with the final element of the chunk). In my examples, the eos signal is sent when the reward>0.9 (in playback mode when the output is fed back to input, feed back also cblockOutputs/eos_predicted), but it can be totally independent of reward – any external signal that makes sense. Self-gen signal should be set high when the input is coming from cblocks own prediction (in a feedback loop) to prevent training on non-sense – see in the next section.

3.3 Learning from self-generated input

Ideally, the baby would explore and learn different motor plans and utilize those that were rewarding. But if the baby's behaviour is already driven by predictions of the seq-SOM, we can arrive at fixed-point behaviour by reinforcing/learning non-sense. That is why (at least initially), training is disabled for inputs that come from own predictions. This can be regulated by cblockParams/seq_learn_from_user vs cblockParams/seq_learn_from_playback for seq-SOM and cblockParams/plan_learn_from_user vs cblockParams/plan_learn_from_playback for plan-SOM.

3.4 Goal setting in goal-driven behavior

A temporary representation of the ongoing motor program is created in the seq-SOM tonic input medium during sequence observation, whether it comes from user, or own playback. The system can also operate in a different – goal-driven – mode in which the tonic input that co-drives the next element prediction comes top-down from the plan-SOM. When cblockInputs/goal_driven=1, the connectivity for the plan-SOM changes so that it finds the best matching motor plan for the difference between the current state and a desired goal state (cblock/planning/goal/state) or reward with their

relative importance determined by cblock/planning/goal/alpha_goal_state_bulk or alpha_reward.[3] Finer-grained attentional weights on individual components of state can be specified in planning/goal/alpha_goal_state_components, thus implementing "neuroeconomics".

The winning plan is sent to seq-SOM's tonic input and stays there till the plan finishes in one of these ways:

- Surprise (if surprise-based boundaries allowed in cblockParams/enable_natural_boundaries) – plan terminated unexpectedly because of mismatch in prediction.
- Predicted eos fed back to input – the plan finished naturally when it was expected to end.
- Good match between the goal (result and/or reward along with their alphas) and the actual result and reward – in this case the plan finishes because the goal has been achieved (you may need to fine-tune cblock/planning/dist_from_goal/activation_sensitivity).
- Timeout – playback got stuck because of high entropy or being too slow/long. The timeout is LIF-driven and depends on cblock/planning/control/plan_timeout_speed. The same timer periodically brings the system into goal selection state (in goal-driven mode) even if no action happens because the goal is reached or there's no good plan. As soon as either the desired or actual state changes, the plan will be updated at the next LIF's pulse.
- User reset plan manually by pressing a button (cblockParams/reset_goal).

During the plan execution, the goal is protected in a goal buffer. That is why if a new goal arrives in the goal medium while a plan is being executed, it is only read in when the ongoing plan finishes. There's a small IOR on previous plan (cblock/planning/control/ior_decay), so that, e.g. in reward-focused behavior the plan-SOM iterates between different plans. The IOR is just on the winning neuron if cblock/planning/control/ior_surrounding=0, otherwise it is a blob around the winner, size of which is regulated by cblock/planning/xy_to_loc_som/activation_sensitivity.

4 Natural chunk boundaries at surprise, chunk length

A safe way of creating chunks is providing an external eos signal at desired chunk boundaries. Cblock is also prepared to automatically detect chunk boundaries at places where the incoming element doesn't match the predicted distribution. At each sequence step, the error/mismatch (or degree of surprise) is measured as KL-divergence between the predicted and actual distributions. Then it is compared to a sliding average of error over time and if it is unusually high, a surprise is detected and a chunk boundary inserted (if cblockParams/enable_natural_boundaries>0). There are several parameters (in cblockParams) influencing this process that you may need to fine-tune:

prev_avg_err_mix – a mixing coefficient for sliding average of error. High values mean slow update, i.e. long time window, small values mean fast update or small time window (extremes: 1= no update, 0 = avg replaced with the most recent error)

surprise_avg_mult – a multiplicative threshold on error – a surprise is detected if the current error is greater than surprise_avg_mult *average

Sometimes we want to limit a maximum sequence length (because if there is no surprise and no external eos, a sequence would never end and its plan would never be learned in the plan-som. You

[3] Depending on the nature of 100-dimensional system state, you may need to fine-tune cblock/plan_som/asomConsts/qmlSensitivityResult (used when alpha_goal_state_bulk>0) and qmlSensitivityJustReward (used otherwise).

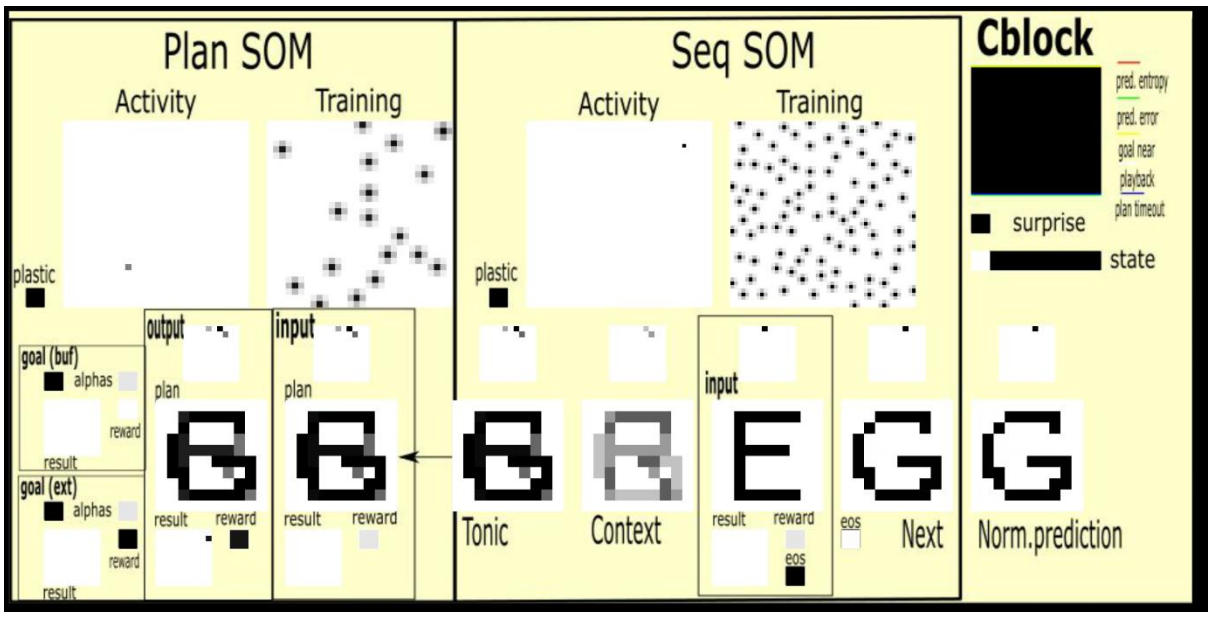
may want to limit the maximum sequence length and force a chunk boundary when reached (param max_seq_length). Also if everything is surprising, plan-som would learn many non-sense 1-element plans. To prevent that, you can set a minimum limit on surprise-based plans (seq-som's context would still be reset at surprise, but the plan-som would not learn shorter plans). Note that this limit doesn't influence external eos based plans – they can be of any length, even 1 (this often happens by mistake fi user sets external eos or reward manually with a slider and then forgets to turn it off right after the asserted final move of the chunk).

5 Example: Letters

This is an example of bitmap chunking, where the system learns sequences of letters as they appear in names of SM employees (see words.txt in the demo's root folder). Each name is a chunk/plan, that can be replayed later. You can type letter sequences manually, but the best way how to try the system is to turn on qml slider Auto-training (in Control section). After a while, when there's no more much change in the training records of the two SOMs, turn it off (If using a keyboard instead, choose a word index as a resulting state (qml control/manual_result_index), input a word letter by letter and set reward to 1 before pressing the last letter of a word, then set reward back to 0).

For goal-driven behaviour: In Goal section, choose some non-zero "result index", then turn on "goal-driven" and "allow playback" in Control section. Watch the "Input" field of seq-SOM visualisation. It should play a name (and then do it again, unless you changed the result index to some other name – this is because during self-generation the state is not updated with the word index at the end of the word as it was during autotraining, hence goal_reached is not detected and the difference to eliminate stays the same while the goal state is the same). Experiment with different names. This was goal-driven behaviour focussed on resulting state (because its alpha in Goal section is set to 1).

Now set the alpha-result to 0 and alpha-reward to 1. Now the system will replay whatever name wins the competition (they were all stored with reward=1 during autotraining). If you set (in Parameters section) "plan IOR decay" to a high non-zero value (close to 1), the system will now generate different names one after another.



(Known issue: sometimes autotraining is finished in a wrong moment and the subsequent goal-driven behaviour is messy – a plan appears in Seq-som's tonic just immediately after the goal reset, then disappears and playback is driven by an empty tonic. This seems to be a bug in letters demo autotraining, not in cblock. If this happens, simply reset the weights and do autotraining again, or restart the whole runtime, or just add a bit more of autotraining.)

6 Drawing example

This is an example of xy position chunking. An element is generated when user puts the pen down or moves the qml widget by more than drawing_params/delta_x_thr or delta_y_thr. Either enable natural boundaries, or set reward manually to 1 with the last stroke (don't forget to turn the reward back off right after it, otherwise there will be a new chunk end after each element while reward=1). See scene/connect_cblock.blc for an example how to connect cblock with xy inputs.

(Known issue: because the qml tab only loads in values when displayed, if using qml cursor widget for drawing, click on the Drawing tab before putting the pen down, otherwise the click itself will generate a spurious dot).

The main difference between the letters and drawing is the nature of result state: in the former, it is a 1hot code for a person's name, in the latter it is a bitmap – drawing. Hence input to Euclidean distance that is at the core of each SOM would be radically different – you might need to fine tune the respective sensitivities differently for each example: planning/dist_from_goal/activation_sensitivity and plan_som/asomConsts/qmlSensitivityPredictResult – see config_cblock.blc in both examples for reasonable values. This example also nicely demonstrates difference-based plan chaining: if let's say a square is drawn in several strokes and each of them is stored as individual chunk during training, then in goal-driven mode with the square as desired state (and starting with erased drawing board), they would be executed sequentially until the whole square has been drawn.

7 Other resources

There's a bunch of cblock-related resources on the CogArch team google drive. These are the most up-to-date (an self-contained) ones:

- Doc/cblock-chunking/cblock.pdf – up-to-date maintained version of 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 some older ones too – it probably still makes sense to look at Doc/cblock-chunking/Chunking.pdf – a document explaining the philosophy of chunking in detail (some parts of it obsolete, but most still valid). Only read if you want to get more background about how it all works

internally. Older videos (I tried to cover most of the stuff in the two videos described about), so these ones are now obsolete:

- Videos/cblock-drawing.mp4 – a video showing how to control the Drawing example described above
- Videos/cblock-letters-manual-training.mp4 – a video showing how to control the Letters example described above (trained manually by typing in letters)
- Videos/cblock-letters-autotraining.mp4 – a video demonstrating autotraining and goal-driven behaviour in the Letters example described above.