A symbolic/subsymbolic interface protocol for cognitive modeling: Supplementary Material

Patrick Simen[a], Thad Polk[b]

[a]Princeton Neuroscience Institute
Princeton University
Princeton, NJ
[b]Department of Psychology
University of Michigan
Ann Arbor, MI

Abstract

In this supplement to the article 'A symbolic/subsymbolic interface protocol for cognitive modeling', we describe the details of a neural network model of the Tower of London task. A similar model has been shown to reproduce the behavior of normal human subjects and also patients with prefrontal brain damage (Polk et al., 2002) and Parkinson's disease (Simen et al., 2004). We discuss in the second section a technique for parameterizing dynamical systems that proved essential to the construction of this model.

1. Problem-space search by a neural production system

In this section, we apply the principles of logic- and architecture-level design that we have presented in previous sections to construct a working model of brain circuits essential for step-by-step problem solving. We apply this to a task that we have previously modeled with a hybrid neural/symbolic model to explain problem-solving impairments by patients with prefrontal brain damage (Polk et al., 2002). The task is the Tower of London task (Shallice, 1982), a puzzle based on the Tower of Hanoi disk-stacking puzzle that was developed for use with potentially brain-damaged subjects. It requires a starting configuration of colored balls on pegs to be transformed into a goal configuration in as few moves as possible (see the bottom of Fig. 1 for a representative problem).

In modeling this task, we confront many of the same difficult choices about how to represent knowledge in a model — and about how *much* knowledge to build into it — that symbolic modelers face in cognitive psychology, and that programmers of expert systems face in artificial intelligence (AI). Such limits fundamentally shape the predictions of any problem-solving model that acquires most of its knowledge from a programmer, rather than from a training process. (Unfortunately, programming a significant amount of knowledge into models of complex cognition is currently unavoidable, both because we do not have a well-understood learning algorithm to acquire the types of model components and connections we use, and because the amount of simulation necessary to train up the relevant task representations under a plausible algorithm would likely pose an insurmountable barrier to modeling complex cognition in practice.)

To see how a psychologically implausible knowledge base can impact a theory of problem solving, consider the extreme case in which all relevant knowledge already exists in a system. To implement such a system in a neural network, a programmer could use the design principles we have presented to encode exhaustively the optimal solution trajectories for all possible problems, identifying each with a unique identifier node, and then causing the system simply to execute an appropriate, pre-existing solution sequence whenever any Tower of London problem is presented. Such a model may be suitable as a model of well-learned sequence performance, but it is highly unintuitive as a model of problem solving (that is, as a model that can produce sequences of actions that it has never before produced). Furthermore, for every problem requiring n moves, it has at least n parameters (the units that encode the moves and the weights that encode the transitions between them in a particular solution), making it extremely unparsimonious.

When the necessary sequence is not already stored, however, it must be created. To do this, the problem space must be explored, meaning that possible solutions must be created and evaluated. Despite the fact that the standard AI approach to problem solving is a sequential search process, we need not accept this approach unquestioningly. Solutions, after all, could conceivably be created and evaluated in parallel. We have already seen how a large number of individual responses can be considered simultaneously in a single, winner-take-all process that appears quite different from the typical decision-making procedure at the architecture-level and below in digital electronics. Why could the same sort of process not simply select a complete solution in one 'step' of parallel processing? In fact, this type of solution may very well characterize the process of solving problems by *insight* (Sternberg, 1999).

Our reason for favoring a sequential, AI-style search model is that, in the type of winner-take-all decision-making process

Email addresses: psimen@princeton.edu (Patrick Simen), tpolk@umich.edu (Thad Polk)

we have examined, we need to allocate one unit for every possible alternative being considered. In order to apply this approach to the selection of an entire sequence of atomic operators, rather than the atomic operators themselves, we must resort to the type of model we have already discussed and rejected — a model with a combinatorially explosive representation scheme in which one node is assigned to every possible combination of operators. This approach is not only inefficient in terms of unit allocation; it also unrealistically assumes that every stored combination has been experienced and memorized. Some other form of parallel, distributed processing could perhaps accomplish problem space exploration without relying on a winner-take-all process, but to our knowledge, no well-developed theory exists yet for how such processing can be applied to problem solving; the processing dynamics of the long-term, subconscious thinking involved in insight are likely to be much more complicated than simply a high-dimensional drift-diffusion process, for example.

Problem-space exploration therefore seems to require sequential processing, at least in some circumstances. With sequential search, a minimal-knowledge approach requires only that the atomic operators be encoded. Then completely random moves can be tried at every decision point. If more knowledge is encoded, then random moves can be relegated to those decision points that involve complete uncertainty. Otherwise, the relevant knowledge about the next move or the next subgoal can be implemented efficiently and deployed to bias action selection only at the relevant decision points during the creation and evaluation of a solution sequence — for example, the knowledge that when a ball cannot immediately go into a desired position, there must be a particular obstacle that the system can work on removing.

We now describe the model in greater detail. Since it consists of a large number of units and connections, we focus primarily on some critical components that illustrate the basic design principles we have described, and that contributed most directly to our explanation of cognitive deficits in patient populations (Polk et al., 2002; Simen et al., 2004). These explanations were based on extensive simulation using a large subset of all possible Tower of London problems.

1.1. Tower of London model

The model is hierarchically organized into two processing levels. The bottom level consists of a sensory-motor decision-making circuit. This circuit abstracts away from the details of object recognition and spatial perception in its sensory interface to the environment, and from the details of motor planning and execution in its motor interface to the environment. This is done for practical rather than theoretical reasons. It allows us — at the risk of being misled by our assumptions (Dreyfus, 1979) — to focus on the interaction between sensory and motor processes in problem solving and how these are modulated by cognitive processes. The environment itself is simply a symbolic representation of the current Tower of London problem configuration which is updated appropriately in response to threshold-crossings of move representations in the action-selection network.

The top layer of the processing hierarchy consists of a goal/subgoal loop circuit that biases the processing in the underlying, feedforward sensory-motor circuit.[1]

Sequential processing is achieved in the model by the approximately punctate, threshold-crossing behavior of strongly self-exciting units, especially those in the model's `Move` layer, despite the fact that the spread of activation from unit to unit is continuous in time in every component of the model (quantum leaps in activation can occur only as the result of white noise, but such leaps occur with probability zero). This type of step-by-step processing takes place simultaneously in parallel processing streams in different parts of the model. However, these streams form part of an overarching, unitary, sequential search process that waits on the longest computation among these parallel streams (or on a timer that limits processing duration) before synchronously triggering the next step of the search. This behavior is achieved by using the action-selection attractor network (labeled `Move` in Fig. 1) as a fundamental processing bottleneck, whose punctate decisions delimit the atomic components of solution sequences.

The structure of the model is illustrated in Fig. 1. In its modular structure, it resembles the models of Schneider and Detweiler (1987) and Feldman and Ballard (1982). However, unlike those models, which allocate complex roles to different types of units, our modules consist of a single type of unit, with self-excitation parameterized to produce an approximately linear layer of input units (with weak self-excitation) feeding forward in a one-to-one connection pattern to a (possibly) highly nonlinear layer of strongly self-exciting units, as in the main article's Fig. 4 and Fig. 11 (in this two-layer structure, our approach is again reminiscent of Schneider and Detweiler, 1987). Output units of some modules may be left as weak self-exciters, so that the modules implement approximately linear transformations of their inputs. Modules are indicated as winner-take-all modules (modules implementing competitive decision-making dynamics in the input layer and feeding forward to a WTA output layer), latches (modules that can preserve symbolic information indefinitely using strongly self-exciting output units), sources (modules that are active by default, and that must be actively inhibited in order to inactivate), delays (modules which impose propagation delay on the signals they transmit), or any combination of these.

1.2. Sensory-motor backbone

Excluding the influence of goals on behavior, the operation of the model is fairly simple. One `Sense` module is devoted to each position on the gameboard. Each module has one unit for representing each of the values 'red', 'green', 'blue' and 'empty'. A simulated environment clamps these modules to the appropriate values to model perception. Each `Sense` module is a latch, guaranteeing nearly binary representations of the

[1] We note that this feedforward assumption merely reflects the degree of abstraction required to make modeling tractable. Motor-sensory feedback and feedback within the perceptual system itself appears to be fundamental to actual brain organization.

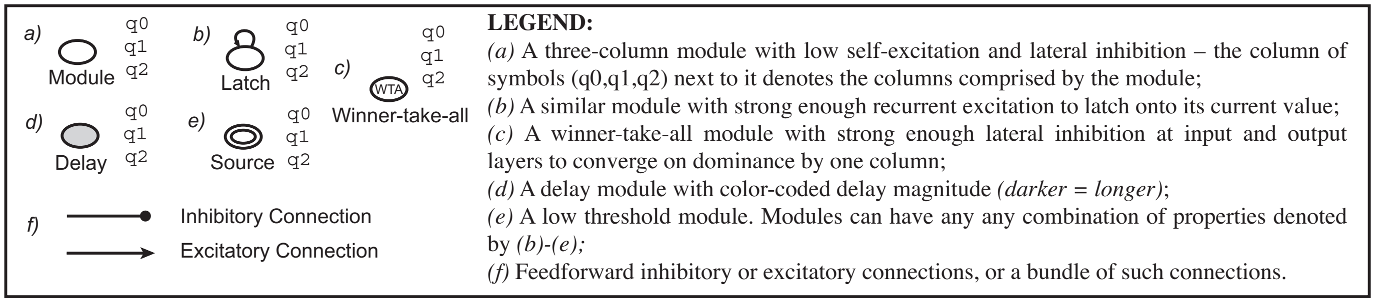

Figure 1: Tower of London schematic. All sets of modules whose connections are not explicitly shown are collections of AND gates that perform combinational logic on their inputs.

color of the ball occupying a given space (this localist representational scheme may represent an abstraction from a pattern of activity distributed across multiple units, and simply makes the model easier to work with). The units in the `Move` module (also a latch) encode possible moves as conjunctions of one of three colors and one of six positions. This 3x6 matrix of move encodings is repeated in several modules in the system. (Note that a permanent, dedicated conjunctive representation of this form is not a scheme that will scale up well to larger problem spaces, as it is wasteful of neural resources. The most obvious solution to this problem appears to be online binding of conjunction components, but this is a form of the variable binding problem and is beyond the scope of this paper.)

The representation of the current configuration (the `Sense` modules) excites all legal moves in the `Move` module with the same degree of preference, and similarly inhibits illegal moves. Constructing connections that encode this excitation and inhibition is a straightforward matter. For example, the unit encoding 'red' in module `Sense1` will excite `Move` units encoding moves of the red ball to any other position on the board. It will also inhibit units encoding moves of any ball to position 1, since that space is occupied. The 'empty' unit in module `Sense4` will vote for moves of any ball to position 4 (in fact, it will do so more strongly than the 'empty' unit in module `Sense3`, in order to bias the system toward moving balls to lower positions on pegs, if possible).

This approach to preference encoding assumes a lack of knowledge on the part of the problem solver about which action should be preferred in any given situation. In an elaboration of the voting process underlying action selection in section 3 of the main article, we could in theory allocate more units and use an automated weight-setting algorithm to ensure that a particular preference ranking among alternatives was encoded for every possible context. Furthermore, we could use such an algorithm to set weights in such a way that integral feedback control of the overall level of activation in the `Move` network was unnecessary. Finally, it is likely that we could do so in a way that met any particular speed-accuracy tradeoff (SAT) criterion, given a particular level of background processing noise (cf. simple methods for controlling SAT in Simen et al., 2006).

Such an omniscient approach to preference ranking, however, seems highly implausible psychologically, and it requires a combinatorially explosive number of units and connections. Instead, we make what we consider to be a plausible, though admittedly *ad hoc* assumption: this is that environmental affordances (such as the perception that a colored ball is within reach) trigger considerations of relevant operators (such as moving the colored ball). Furthermore, since there is no way to know ahead of time how many affordances are liable to influence decision making about actions in any given context, we must employ feedback control to allow for net inputs consisting of a weighted sum of arbitrarily many components, and therefore an arbitrarily large or small vector magnitude. As for our use of selective, inhibitory influences on decision making, we once again note that without them, it would be difficult to encode the limited proposition simply that certain moves should not be considered. This proposition is fundamentally different than the proposition that some *other* move *should* be considered.

1.3. Perceptual reasoning

Logic gates like the one in Fig. 10 of the main article are used to determine which balls are blocked from moving by balls stacked on top of them. This information requires input from two `Sense` modules: the `Sense` module corresponding to the ball that may be blocked, and the module corresponding to the position above it. Thus for each of the blockable positions 2, 4 and 5, there is one module for each of the three colors. For example, if `Sense2` represents 'red', then the `Red2Block` module will be activated to indicate that the red ball may be blocked in position 2. This module excites the `RedBlocked` module, but not enough to activate it. The 'green' and 'blue' units in `Sense3` also excite `RedBlocked`. `RedBlocked` is a neural AND gate that only responds to the conjunction of `Sense2` = 'red' and (`Sense3` = 'green' or `Sense3` = 'blue'). Moves of blocked balls are strongly inhibited by the `Blocked` modules.

1.4. Move selection and gating

Our approach to move selection assumes that low-level rules about what can be moved, and where, have been compiled through previous experience with analogous, real-world stacking tasks. The possible legal moves then compete with each other in the `Move` module via attractor dynamics until one is selected. Without other sources of input, the move that is finally selected is random and simply depends on noise. The `ActivationRegulator` module also supplies diffuse excitation to the `Move` module as part of a feedback mechanism that ensures the winner-take-all property of the `Move` module (that is, if no winner emerges, the `ActivationRegulator` boosts all `Move` unit excitations, but it does so by ramping up slowly enough to ensure that the runner-up representation does not also activate along with the winner; a sufficiently slow rate was determined by trial and error during model construction in Matlab).

Once a move is selected, it excites a corresponding unit in the `MoveGate` module, which is not a latch. The activation of this module can be thought of as a motor command issued to the peripheral motor system. It also serves to extinguish the move selected in the `Move` module, in order to allow for later move elections to take place, as well as serving to extinguish other short-term memories related to selecting subgoals that are no longer relevant. Once `MoveGate` is active above a threshold activation value of 0.9, the simulated environment is updated, causing the representation of the configuration to change accordingly, and the new configuration once again votes for any legal moves. In short, in the absence of goal direction, the model simply performs random search using any moves that are legal in the current configuration. (Another major simplification of our model is that it simply executes actions, without memorizing an internally simulated sequence first.)

1.5. Goals

Goals as they are represented in the system correspond to placing individual balls in specific locations (e.g., getting the blue ball to the bottom of the third peg). Thus goals have the same representational format as moves. The base level goals of the system are represented in the latch modules `Goal1` through `Goal6`. The goal configuration of the gameboard is represented there with six latches, as in the `Sense` system, by initializing each to the color of the ball that occupies the corresponding position.

Only one component of this goal configuration — the *controlling goal* — is worked on at a time, because the move-selection process would become disorganized otherwise (theoretically, though, there is no reason to suppose that multiple, parallel goal-driven processes could operate simultaneously as long as they did not conflict with each other). The identity of the controlling goal is decided by attractor dynamics within the winner-take-all, latch module, `GoalDecide`. Here, weights on the connections from the `Goal` modules to `GoalDecide` were easy to set with a fixed preference ordering that guaranteed a unique winner at all times. The preference scheme favors moving balls to lower positions on pegs than to higher positions, since a ball moved to its final position at the bottom of a peg will never have to be moved thereafter to achieve a complete solution (this choice represents an *ad hoc* insertion of knowledge that is quite conceivably not shared by all subjects, but which seems likely to emerge, at least in healthy subjects, after experiencing a few difficult Tower of London problems). The output of `GoalDecide` is fed to `GoalGate`, which attempts in turn to guide move selection when the system is not already working on some other goal, as discussed next.

In the hybrid model discussed in Polk et al. (2002), `GoalGate` also has connections that can directly influence move selection as well. These connections embody the notion that base-level goals typically guide behavior whenever possible without the participation of the subgoal circuitry that we discuss next (and which might be implemented in the brain dynamically, using resources that would otherwise best be left available for other purposes). These connections provide one explanation for the effects of problem-solving deficits in prefrontal patients as stemming from overly greedy search, which also occurs in the neural Tower of London model in Dehaene and Changeux (1997) (that model does not explicitly model a production system).

1.6. Subgoals

The current controlling goal, if any exists, is represented in the winner-take-all, latch module labeled `Subgoal` (the name derives from the fact that this module also represents subgoals generated by specific subgoal-generation knowledge or by impasses in the problem solving process). The `Subgoal` module modulates processing in the `Move` module by inhibiting all moves while simultaneously exciting the move that could achieve the controlling goal if legal. The net effect is that one move is excited while all others are inhibited. This modulation biases the competition in the `Move` module so that the move that could achieve the controlling goal (if legal) will tend to be selected.

If no legal move will achieve the controlling goal, then with high probability, no move is selected (because the controlling goal will inhibit all legal moves in that case). We assume that the model may decide either to select some move randomly, or to remain below threshold throughout the duration of an interval timed by an interval timer, at which point the system detects that it has reached an impasse in its search. Then a new subgoal can be generated, or work can commence on a different base-level goal.

Once again, however, incorporating more knowledge into the system would make this approach unnecessary. If the system incorporated enough knowledge to detect when no move is possible in all possible contexts, then it could rely on a 'no-move' alternative in addition to the 18 possible moves in the `Move` network. It could then dispense with the interval timer and use 'no-move' as a default output in the absence of strong enough excitation of alternative moves. In a task as simple as the Tower of London, it seems plausible that such knowledge might be available to a subject after solving many instances of the task. However, we are interested in the Tower of London task only as a means of testing an architectural template that we hope will be useful as a component of general theories of problem solving. Therefore, assuming minimal knowledge during search in an architecture based on decision-making mechanisms that produce arbitrarily long decision times, and where impasses are inevitable, seems to require interval timing as an integral component. Such impasses in Soar, for example, trigger the generation of new search spaces. In these spaces, the solution to an impasse in a given context is computed by a separate exploration process. The solution sequence is then chunked into a new operator that prevents impasses in future instances of the same context.

In order to make our modeling effort tractable, however, we do not model search space generation. Instead, we assume that such meta-searches have already occurred, and we pre-chunk their results into knowledge that is relevant to the current controlling goal. We use latch modules to represent what is above the ball to which the goal refers (`AboveSource`), the color of the ball, if any, in the target position (`InTarget`), and the lowest free position on the peg that is neither the source nor the target of the current goal (`FreePosition`). This information is crucial for generating the right subgoals to get blocking balls out of the way without disrupting progress toward the controlling goal. It is computed in the same manner that blockage information is computed by the Sensory system, by using a cascade of logic gates that receive input from the `Subgoal` module and the `Sense` modules. Connections from these modules to the `Subgoal` module encode the knowledge about which subgoals to try that we assume has been acquired by impasse-generated meta-searches, or else simple trial and error.

The selection of a new goal in the `Subgoal` module can occur in one of two ways. First, if the current goal has been achieved, then it is inhibited by the `Sense` modules (e.g., `Sense1` = 'red' inhibits the goal 'Red to 1'). Then one of the still-unachieved base level goals is retrieved from the `GoalGate` module through

a handshake procedure. The handshake works as follows. All units in Subgoal inhibit all units in GoalGate. Because of this inhibition, GoalGate cannot vote for Subgoal to instantiate a base-level goal once progress has begun on some other goal — that is, while Subgoal is actively representing something. (This is a case in which knowing how to parameterize a closed-loop, self-cancelling production is critical for achieving proper function.) Once Subgoal has reached baseline activation in all units, however, GoalGate is able to become active for long enough to install a new base level goal in Subgoal. The handshake is completed by Subgoal effectively issuing an 'all done' signal to GoalGate.

The second way in which a new goal can be instantiated in Subgoal occurs when the current goal cannot be directly achieved because of some obstruction (either a ball in the target position or a ball above the ball that we want to move). Then a new goal to remove the obstruction will be proposed via strong input from Abovesource, InTarget, and FreePosition that overwhelms any input from GoalGate. In order to compute a new subgoal in this way, the current output of Subgoal must not be destroyed prior to the computation of the new subgoal. This output is fed into a logic circuit that feeds back into the input of Subgoal, posing the risk of a critical race condition, which flip-flops prevent.

We have discussed the use of a goal stack, which was in fact implemented in earlier iterations of the hybrid neural/symbolic Tower of London model in Polk et al. (2002). However, once subgoals were achieved in these earlier versions of the model, it was frequently the case that evaluating the problem from the current configuration in relation to base-level goals, rather than in relation to a parent subgoal, provided more flexible performance. For example, if a parent subgoal was to remove a blocking ball so that the green ball could be moved to its goal position, it often turned out that it was now serendipitously possible to move the red ball to a more secure target position (a position lower down on a taller peg). By allowing the model to rebuild its goal stack after every move, it was better able to capitalize on such opportunities. In this respect, the model acts more like recent versions of ACT-R, with its flexible, activation-based goal organization, rather than a rigid stack organization.

1.7. Impasse detection

In cases in which the Subgoal module is inhibiting moves that would not help achieve the current goal, connection strengths are such that the Move module will never select a winner. In this case, a timer circuit will be triggered by a lack of activation in the Move module. This is the purpose of the NoMove timer chain in Fig. 1.

The NoMove timer chain starts with a single-unit source module that has very small β in Eq. 7 of the main article, so that it will tend to activate without external inhibition. Because of finely tuned recurrent self-excitation, each element in the chain ramps up slowly and nearly linearly. When the last element in the chain finally activates, it prevents any move from emerging by strongly inhibiting the Move module and begins a process for generating a subgoal. (Technically, one NoMove unit would have sufficed, but it was easier in practice to parameterize a chain of them.) The purpose of this inhibition is to make the decision to generate a subgoal irrevocable. This stops any later Move activation that would disrupt the subgoal generation process, or cause moves in the meantime that would make the ultimately generated subgoal inappropriate for the resulting configuration of the gameboard.

Once activation reaches the final timer module in the NoMove timer chain, the Generate module is activated. This module allows information about the current goal to flow through SubgoalBufferGate, at the same time choking off information about the current goal from flowing through SubgoalGate. The purpose of this is to prevent a critical race condition. The Generate signal is thus the analogue of a single clock pulse in a synchronous digital circuit, triggering the neural flip-flop formed by the following chain of modules: Subgoal, SubgoalGate, SubgoalBuffer, SubgoalBufferGate. At this point, a cascade of logic gates uses information about the current subgoal and the current Sense configuration to determine which ball, if any, is above the ball to be moved under the current goal, which ball is in the target position of the ball to be moved, and which peg is able to hold moved blockers. These three values then vote for appropriate subgoals in the Subgoal module (i.e., to move to the free position any target-position blocker, if present, and if not, then the source-ball blocker).

The successful generation of a subgoal will be signalled by the elimination of the old Subgoal pattern, followed by the activation of a new one. This process is detected in the sequence of modules Compare and GenerateSuccess. Compare detects activation in Subgoal, but only after it has been enabled by CompareEnable. CompareEnable itself cannot be active until the first goal pattern in Subgoal has been extinguished. This subgoal generation process is ended as soon as a successful generation is detected by Compare, which simply responds to any significant activation in the Subgoal module. If desired, we could implement a second subgoal-selection timer, since attempting to push a subgoal might itself generate an impasse. Expiration of this timer would indicate that there is no appropriate subgoal to generate. In this case, one of the search heuristics implemented in Polk et al. (2002) could be used to select a subgoal according to different logic than that encoded in the logic-gate cascade, or a meta-level search could be initiated, as in Soar. For the sake of limiting model complexity, however, we focus on an extremely simple model that is capable of solving most Tower of London problems requiring five moves or less (as is the case, to the best of our knowledge, in all empirical studies involving performance by prefrontal patients and Parkinson's patients of more difficult problems).

1.8. Implemented algorithm

The following algorithm that the neural Tower of London model carries out is a simplified form of that implemented in the hybrid system in Polk et al. (2002), which used extra heuristics to deal with difficult cases. The position numbers referred to in the algorithm are illustrated in the goal configuration at the bottom of Fig. 1. 'Position 1' refers to the only space on the shortest peg, 2 denotes the bottom of the medium-height peg, 3

IF	THEN	Strength
CurrentConfig Pos1 = *red*	**Move** = *red_to_2*	*0.2*
CurrentConfig Pos1 = *red*	**Move** = *red_to_3*	*0.2*
CurrentConfig Pos1 = *red*	**Move** = *red_to_4*	*0.2*
CurrentConfig Pos1 = *red*	**Move** = *red_to_5*	*0.2*
CurrentConfig Pos1 = *red*	**Move** = *red_to_6*	*0.2*
CurrentConfig Pos1 = *red*	**Move** = *green_to_1*	*-1.0*
CurrentConfig Pos1 = *red*	**Move** = *blue_to_1*	*-1.0*
CurrentConfig Pos1 = *red*	**Move** = *red_to_1*	*-1.0*
	⋮	
CurrentConfig Pos2 = *green* Pos2 = *blue*	**PerceptualLogic** Green2Block = *on*	*1.0*
	⋮	
PerceptualLogic Green2Block = *on*	**PerceptualLogic** GreenBlocked = *on*	*1.0*
	⋮	
PerceptualLogic Green2Block = *on*	**PerceptualLogic** GreenBlocked = *on*	*1.0*
	⋮	
PerceptualLogic GreenBlocked = *on*	**Move** = *green_to_1*	*-1.0*

IF	THEN	Strength
Goals Pos1 = *red*	**GoalDecide** = *red_to_1*	*0.1*
Goals Pos2 = *red*	**GoalDecide** = *red_to_2*	*0.2*
	⋮	
AboveSource = *green*	**Subgoal** = *green_to_1*	*0.1*
	⋮	
AboveSource = *green*	**Subgoal** = *green_to_6*	*0.1*
	⋮	
InTarget = *green*	**Subgoal** = *green_to_1*	*0.1*
	⋮	
InTarget = *green*	**Subgoal** = *green_to_6*	*0.1*
	⋮	
FreePosition = *3*	**Subgoal** = *red_to_3*	*0.2*
FreePosition = *3*	**Subgoal** = *green_to_3*	*0.2*
FreePosition = *3*	**Subgoal** = *blue_to_3*	*0.2*
	⋮	
Subgoal = *red_to_3*	**Move** = *red_to_3*	*0.2*

Table 1: A subset of the productions implemented by the neural Tower of London model. Those shown illustrate the basic computation of legal moves; perceptual relationships (which balls are blocked); current goals; and subgoals in the case that an impasse is reached.

denotes the space on top of 2, 4 denotes the bottom of the tall peg, 5 denotes the space above 4, and 6 denotes the space above 5. The algorithm is as follows:

1. set a goal to move a ball to its final goal position (with preference, in decreasing order, for moves to position 4, then 5, then 2, then 6, then 3, then 1)
2. if the current goal is not achieved because some ball is blocking the ball to be moved, set a subgoal to move the blocking ball to the lowest position on the peg which is neither the source nor the target of the current goal — if there are two such pegs, pick one at random — then return to step 2; otherwise, make the desired move;
3. as soon as a move is made, return to step 1.

Table 1 illustrates a set of symbolic productions that the neural model implements. Only a small subset of the total number of productions are shown. All rules match in parallel, and the strength factors in the righthand column determine which ones fire when there is conflict between the outcomes of multiple rules.

1.9. Performance of the model

Here we illustrate the performance of the model in one problem, simulated without noise for clarity's sake, using a Runge-Kutta(4,5) ordinary differential equation solver implemented in Matlab. Timecourses of activation in most of the model's components are shown in Fig. 2. The problem, shown at the bottom of Fig. 1, requires five moves for solution and therefore requires that some balls be moved to positions other than their final, goal positions. Thus it requires the internal generation of subgoals for efficient solution.

The `GoalDecide` module can be seen to hold an election for the first goal to control behavior. The goal 'Red to 4' wins at approximately time 25 (labeled A in the figure — time units are arbitrary; a fit to behavior would be required to define them, but we speculate that milliseconds would be appropriate). 'Red in 4' is the most important unachieved goal according to the preference scheme for moving balls to the lowest possible positions. This information is then transmitted through the `GoalGate` module to the `Subgoal` module, which responds to it at label B. (In the meantime, in order to prevent the premature generation of a subgoal in response to `Move` module inactivity, the `Start` module inhibits the `NoMove` timer system.) The `Sense` modules, like the `Goal` modules, are initialized at the beginning of the simulation and excite potentially legal moves at the same time as the `Blocked` modules compute which balls are blocked. Finally, a winner, 'Red to 4' is selected at time point C, and the corresponding unit in `MoveGate` is caused to rise to threshold, achieving the move and wiping out the move-generating command in `Move`. At this point, the simulated environment causes an update of the `Sense` modules (point D). These in turn extinguish any goal or subgoal activation patterns in the `Goal` system or in `Subgoal` which represent goals to create the already achieved environmental configuration (point E). This allows the next most preferred goal to be retrieved and worked on, as can be seen in `Subgoal` at point F. At no point is the local clock circuit `NoMove` involved.

Now the next goal, 'Blue to 2' is selected. This is unachievable, however, and causes an impasse that times out the `NoMove` timer. This in turn generates a subgoal to remove an obstacle (we trace the timecourses of a subgoal generation process at the next impasse that will occur). Once a subgoal is selected ('Green to 5', since Green is in the target position of the blue ball and position 5 is free, at time point G), the first element of the `NoMove` timer sequence begins to ramp up. When maximal activation finally reaches the last timer in the sequence, the system times out a second time at time H. At this point, the `Generate` module begins a new subgoal generation process. This in turn enables testing for subgoal-generation success by activating `CompareEnable`, while also allowing the information about the current goal to filter into the subgoal computation modules through `SubgoalBuffGate`, at time I. Activation in `Generate` chokes off `SubgoalGate` so that information about the new subgoal cannot propagate until the next generation process, at time J. It also wipes out the current `Subgoal` pattern. Finally, the subgoal generation logic computes that the ball above the green source ball is blue, at time K, and that the

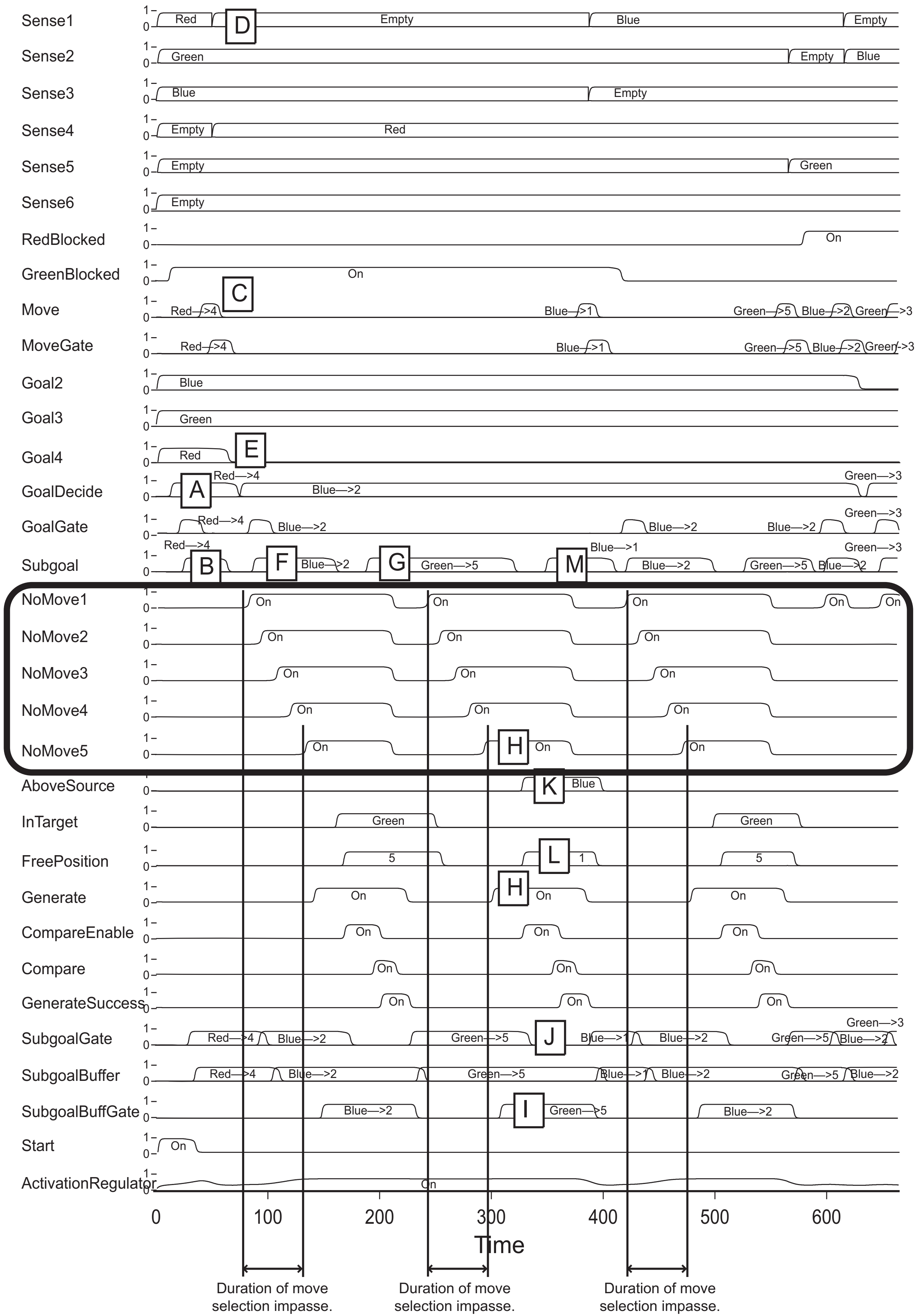

Figure 2: Time courses of activation in most modules of the neural Tower of London problem solver, with noise coefficients reduced to 0 for the purpose of illustration. The activations of multi-valued symbolic attributes are labeled with the name of the current value, while binary-valued attributes are labeled with 'On'.

lowest position on a peg which is neither the source nor the target of the goal is position 1 at time L. `Subgoal` responds to this voting at time M with a goal to move blue to position 1. The model continues on in this way until eventually solving the problem in 5 moves, as shown in the sequence of gameboard configurations that appear in the `Move` module trace: 'red to 4', 'blue to 1', 'green to 5', 'blue to 2', 'green to 3'.

Averaged model performance over many different game configurations is shown for similar models in Polk et al. (2002) and Simen et al. (2004). Matlab code for this model is available at: http://www.math.princeton.edu/~psimen/SimenPolk_TOL_Code.tar.gz.

2. Cobweb diagram approach

In this section, we examine the mathematical details behind a useful technique for neural cognitive model construction.

When examined closely, the vector field arrow-plot in Fig. 3 shows that upward velocities for input values greater than A in Fig. 6 of the main article are extremely small if the system output was recently near 0 (i.e., if it was near the lower stable-equilibrium curve). By definition, they go to 0 on the curve itself, and they change continuously as a function of position in the graph. Thus there is a region near $(A, 0)$ in Fig. 6 of the main article where the system will move upward, but extremely slowly. However, it is difficult to develop any sense, from looking at the vector field, of how slowly it moves, nor of how slight changes in the input strength affect this speed: the arrows are too tiny. Speed, however, is the property that we would like to control.

We can arguably get a better sense of the speed at which the system is changing as a function of current input value and current output value by using cobweb diagrams (Jordan and Smith, 1999). Cobweb diagrams are typically used to determine how discrete-time difference equations evolve over a sequence of time steps. A simple example of a difference equation is given in Eq. 1:

$$y_{n+1} = f(y_n). \tag{1}$$

Cobweb diagrams that are equivalent to the phase-plane plots in Fig. 7 of the main article are shown in Fig. 3.

Eq. 1 determines the evolution of a variable y by setting the new value of y at time $n + 1$ equal to $f(y_n)$. If f is nonlinear, it may be difficult to determine analytically what the behavior of y_n will be. Cobweb diagrams provide an efficient method for visualizing this behavior. In a cobweb diagram, the x-axis represents the value of y at time step n, and the y-axis represents y_{n+1}. We can determine its value by plotting $f(y_n)$ as a function of y_n. A 45° line through the origin is then used to transfer the value $y_{n+1} = f(y_n)$ from the vertical axis to the horizontal axis, so that y_{n+2} can be computed, and the process can repeat. This transfer involves tracing from $f(y_n)$ horizontally to the 45° reference line, and then vertically down to the horizontal axis. Evaluating $f(y_{n+1})$ at the new position on the horizontal axis is then equivalent to tracing vertically back up to f. To save on tracing, we compress the last two steps, tracing from $f(y_n)$ across to the reference line, and then up or down to f. The effect of applying this algorithm is to trace out a characteristic

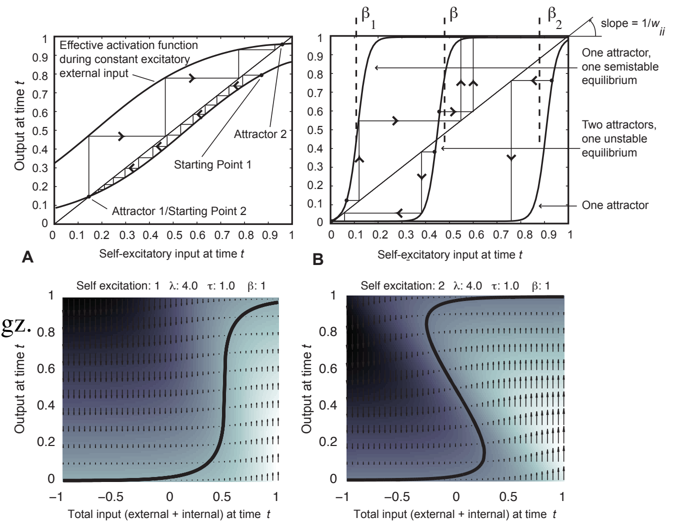

Figure 3: **Top row:** Cobweb diagrams for units with balanced (A) and strong (B) recurrent connections to themselves. The rate at which a unit's activation approaches the nearest attractor is determined by the height of the stair steps depicted. The shift of the activation function equals the weighted sum of inputs from other units. (See appendix 2 on analyzing nonlinear dynamics with cobweb diagrams.) **Bottom row:** equivalent phase plots of the system's velocity for parameterizations identical to the top row, with equilibrium curves plotted as solid curves, and velocities plotted by arrows and shading (white corresponds to positive, black to negative velocities).

sequence of stair-steps (or a cobweb) between f and the reference line.

In our case, however, the units of Eq. 24 in the main article are defined in continuous time by differential rather than difference equations. Nevertheless, as we now demonstrate, the dynamics of self-exciting units receiving *constant* inputs from other units are accurately characterized by the self-excitation cobweb diagrams of Fig. 3 (cf. Harth et al., 1970; Seung et al., 2000; Wilson and Cowan, 1972). When interpreting these diagrams, we temporarily consider the continuous-time system defined by $dV/dt = -V + f(\text{input})$ to be a discrete-time difference equation (Eq. 2):

$$V_{n+1} = f(\text{input}). \tag{2}$$

This temporary assumption is equivalent to using Euler's method for approximating the derivative of a function, $dx/dt \approx [x(t + \Delta) - x(t)]/\Delta$, or $x(t + \Delta) = x(t) + dx/dt \cdot \Delta$, with $\Delta = 1$:

$$\begin{aligned} V(t+1) &= V(t) + dV/dt \cdot 1 \\ &= V(t) + (-V(t) + f(\text{input})) \\ &= f(\text{input}). \end{aligned} \tag{3}$$

Of course, setting Δ to 1 typically causes Euler's method to produce a terrible approximation of $x(t + \Delta)$. It is therefore important to note that we are *not* making this discrete-time assumption in order to approximate $V(t + 1)$. We are doing it in order to relate the height of the stair steps in a cobweb diagram to the velocity of the continuous-time system at any point in the x, y-plane. This relationship underlies a simple, graphical

method for comparing the rates of change of a unit's activation under different levels of fixed input. All we will have to do is count steps: wherever many small steps occur, we can be sure that the system will change slowly; wherever a small number of large steps occur, the system is guaranteed to change quickly. Cobweb diagrams will therefore illustrate how we achieve control over latching speed and how interval timing can be achieved with nonlinear units.

Before we can fully exploit cobweb diagrams however, it will be useful to define an *effective* activation function, which is an activation function that is shifted by the amount of input to a unit from other units. If a unit's activation function is $f(x)$, where f is a logistic sigmoid function with particular values for β and λ (Eq. 24 in the body of the article), then we define the effective activation function $f_\delta(x) \equiv f(x+\delta)$. This change of variables allows the input to a unit through its recurrent, self-excitatory connection (corresponding to x) to be separated from the weighted sum of inputs from other units (corresponding to δ). We refer to δ as the *external input* to the unit, and x as the *recurrent input*.

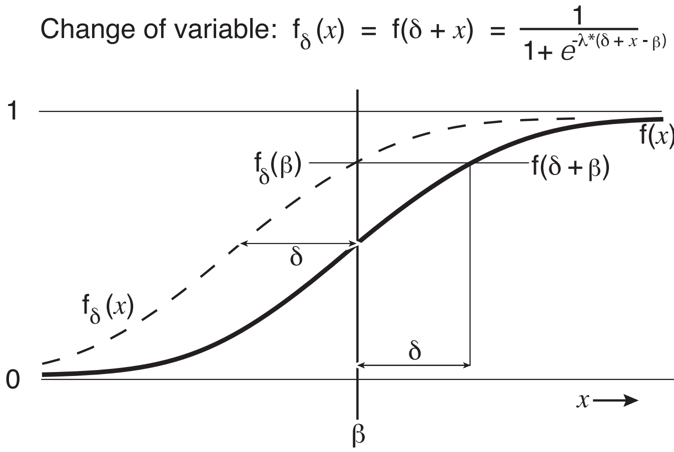

Figure 4: Left-shifted effective activation function f_δ (dashed curve) resulting from an external input of magnitude $+\delta$ to a unit.

Now we apply cobweb diagrams and variable changes to the analysis of self-exciting unit dynamics. In plot A of Fig. 3, two different activation curves are shown. Each corresponds to a different level of external input, in accordance with the following principle: *net excitation δ results in a leftward shift of the activation function by δ; net inhibition $-\delta$ results in a rightward shift of δ* (see Fig. 4). A position on the horizontal axis therefore represents only the recurrent input.

Assuming that the system begins with some output value V_0 and that the weighted sum of inputs from other units is held constant at δ, then we begin tracing out the stair-step trajectory by starting at a point at height V_0 on the vertical axis. We trace horizontally to the reference line to find the input to the unit from itself on the next 'time step' (making our temporary assumption of discretized time). However, unlike in Eq. 1, the reference must account for the fact that the recurrent connection strength may be different from 1, and therefore the input to a unit from itself may require a reference line with slope greater or less than 45°. If the connection strength is 2, for example, the reference line should map V_0 on the vertical axis to $2 \cdot V_0$ on the horizontal axis. Generally, then, if the recurrent connection strength is w, the reference line has slope $1/w$, relative to the horizontal axis.

Now we trace vertically from the input at time step $n = 1$, which is $w \cdot V_0$, to $f_\delta(w \cdot V_0)$ in order to get V_1. Then we trace horizontally again to the reference line to find $w \cdot V_1$, and vertically to get $V_2 = f_\delta(w \cdot V_1)$. By repeating this process, the stair-step trajectory clearly converges to the correct attracting value for V (the same value to which the continuous-time system converges). However, the discrete-time assumption appears to make the *rate* at which the attractor is approached in continuous time difficult to ascertain by this method.

We can address this apparent difficulty by the following geometric argument. As with any system of autonomous differential equations, when a small enough region in the diagrams is considered, the rate of change of V is nearly constant within that region. In our case, such regions are usually large enough to encompass several contiguous stair-steps; this follows from the fact that a given stair-step height is nearly constant when the horizontal position at which the step begins is shifted laterally, within a range of several step-widths. Therefore, the time taken by V to travel the distance D is approximately $D/(dV/dt) = D/\text{step height}$.[2] This means that the number of steps taken to cover the vertical distance D determines (approximately) the time needed for the continuous-time system to cover the same distance.

The only case in which the stair-step heights change rapidly over a small horizontal range is in plot B of Fig. 3. There, sudden increases in the stair step height occur for input levels that shift the sigmoid so that it is nearly tangent to the reference line (these are input levels that are important for our discussion of latching). In these cases, a single step can take the discrete-time system nearly to its attractor from a range of different vertical starting points. Since different starting points necessarily imply different durations of travel to get within some small distance ϵ from the attractor, the step-counting method for computing transit-time becomes inaccurate for very large steps (i.e., with height on the order of 1). This is essentially a form of roundoff error. However, for our purposes, any step that is large enough indicates activation change that is so rapid that it can be considered effectively instantaneous. We can therefore assume that the number of steps in a trajectory fairly accurately represents the time required for the continuous-time system to cover a given vertical distance. Thus we can safely infer the transit-time of a continuously evolving activation simply by counting steps.

We can now assess the effects of different levels of input and different strengths of recurrent self-excitation on a unit's dynamics. Fig. 3 contrasts two different parameterizations of a self exciting unit. In plot A, the recurrent weight is small relative to the slope of the activation function at its midpoint (which is $\lambda/4$). In plot B, the weight is large relative to this slope. The

[2] If the continuous-time system has a time constant τ, then this becomes $(D\tau)/\text{step height}$).

cobweb diagrams in these plots show that weak and strong self-excitation, respectively, result in qualitatively different behavior.

With weak self-excitation, activation exponentially approaches an equilbrium value, so that the unit continues to act like a low-pass filter. With 'balanced' self-excitation that perfectly compensates for the unit's leak, activation that ramps nearly linearly up or down at a controllable rate is possible (depicted in plot A). With strong self-excitation, depicted in plot B, hysteresis occurs: approximately all-or-none activation levels and memory in the form of reverberating activation are possible. Furthermore, we can create arbitrarily small bottlenecks between the sigmoid and the reference line in order to produce arbitrarily slow changes in activation, thereby effectively changing the time constant of Eq. 24 of the main article. Finding bottlenecks that are small enough to achieve slow latching is the key to building models with closed-loop, self-cancelling connection patterns. Finally, when recurrent excitation precisely equals $\lambda/4$, a unit becomes a perfect integrator of its inputs. Thus a constant input δ will produce a linear increase in activation whose slope depends on δ. This provides the basis for our approach to interval timing, in which a drift-diffusion process rises to a threshold value on average at a time T after starting at time 0. Here T = threshold/drift, drift is determined by δ, and the threshold is fixed (Simen, 2008; Simen and Balci, in review).

For a strongly self-exciting unit, the connection strength and the bias term β together define the level of input activation that will trigger a threshold-crossing detection response. If the level of excitation required to shift a strongly self-exciting unit's activation function past the leftmost bifurcation point is A, and the desired activation of the input unit that should trigger a threshold crossing detection is $y_0 < 1$, then the weight w between the input unit and the threshold unit should be $w = A/y_0$. Another important point is that if the input level is y, then the threshold applied to y that is defined by a given weight w is A/w. (This is true for a deterministic version of the system — with noise, simulations suggest that threshold values are approximately normally distributed about these values; this may therefore be considered to be a neural network implementation of the normally distributed threshold model of response times in Grice, 1972).

Finally, we note that self-excitation diagrams represent the continuous-time dynamics of a unit accurately only when its inputs from other units are not changing too rapidly. If inputs are always rapidly changing, then a weakly self-exciting unit will continue to act as a low-pass filter, though with increased gain at low frequencies and a smaller upper frequency cutoff than it would have without self-excitation. A strongly self-exciting unit, on the other hand, will react more unpredictably to a net input that has strong, high-frequency components. In such cases, self-excitation diagrams are not particularly useful.

References

Dehaene, S., Changeux, J., 1997. A hierarchical neuronal network for planning behavior. Proceedings of the National Academy of Sciences, USA 94, 13293–13298.

Dreyfus, H., 1979. What Computers Can't Do: The Limits of Artificial Intelligence, 2nd Edition. Harper & Row.

Feldman, J. A., Ballard, D. H., 1982. Connectionist models and their properties. Cognitive Science 6, 205–254.

Grice, G. R., 1972. Application of a variable criterion model to auditory reaction time as a function of the type of catch trial. Perception and Psychophysics 102, 103–107.

Harth, E. M., Csermely, T. J., Beek, B., Lindsay, R. D., 1970. Brain functions and neural dynamics. Journal of Theoretical Biology 26, 93–120.

Jordan, D. W., Smith, P., 1999. Nonlinear Ordinary Differential Equations, 3rd Edition. Oxford University Press, New York, NY.

Polk, T. A., Simen, P. A., Lewis, R. L., Freedman, E. G., 2002. A computational approach to control in complex cognition. Cognitive Brain Research 15 (1), 71–83.

Schneider, W., Detweiler, M., 1987. A connectionist/control architecture for working memory. In: The Psychology of Learning: Advances in Research and Theory. Vol. 21. Academic Press, San Diego, CA, pp. 53–119.

Seung, H. S., Lee, D. D., Reis, B. Y., Tank, D. W., 2000. The autapse: a simple illustration of short-term analog memory storage by tuned synaptic feedback. Journal of Computational Neuroscience 9, 171–185.

Shallice, T., 1982. Specific impairments in planning. Philosophical Transactions of the Royal Society of London, Series B 298, 199–209.

Simen, P. A., 2008. Ramping, ramping everywhere: an overlooked model of interval timing. In: COSYNE Abstracts.

Simen, P. A., Balci, F., in review. Adaptive interval timing by a noisy integrate-and-fire model.

Simen, P. A., Cohen, J. D., Holmes, P., 2006. Rapid decision threshold modulation by reward rate in a neural network. Neural Networks 19, 1013–1026.

Simen, P. A., Polk, T. A., Lewis, R. L., Freedman, E., 2004. A computational account of latency impairments in problem solving by Parkinson's patients. In: Proceedings of the International Conference on Cognitive Modeling. pp. 273–279.

Sternberg, R. J., 1999. Cognitive Psychology, 2nd Edition. Harcourt Brace, New York.

Wilson, H. R., Cowan, J. D., 1972. Excitatory and inhibitory interactions in localized populations of model neurons. Biophysical Journal 12, 1–24.