A continuous time neural model for sequential action: Supplemental Information*

George Kachergis[1], Dean Wyatte[2], Randall C. O'Reilly[2],
Roy de Kleijn[1], and Bernhard Hommel[1]
george.kachergis@gmail.com
[1] Institute for Psychological Research &
Leiden Institute for Brain and Cognition, Leiden University
[2] Department of Psychology and Neuroscience,
University of Colorado Boulder

1 Leabra and LeabraTI: Implementation Details

The Leabra framework is described in detail in O'Reilly and Munakata (2000); O'Reilly, Munakata, Frank, Hazy, and Contributors (2013) and O'Reilly (2001), and summarized here. The standard Leabra equations have been used to simulate over 40 different models in O'Reilly and Munakata (2000) and a number of other research models. Thus, the model can be viewed as an instantiation of a systematic modeling framework using standardized mechanisms, instead of constructing new mechanisms for each model.

This version of Leabra contains an extension called LeabraTI (Temporal Integration) that allows learning to operate over temporally contiguous input sequences. The full treatment of LeabraTI is presented in an in-preparation paper (O'Reilly, Wyatte, Rohrlich, & Herd, in preparation), but the basic equations and a brief motivation for them are presented here.

1.1 Point Neuron Activation Function

Leabra uses a *point neuron* activation function that models the electrophysiological properties of real neurons, while simplifying their geometry to a single point. This function is nearly as simple computationally as the standard sigmoidal activation function, but the more biologically-based implementation makes it considerably easier to model inhibitory competition, as described below. Further, using this function enables cognitive models to be more easily related to more physiologically detailed simulations, thereby facilitating bridge-building between biology and cognition. We use nor-

*doi: 10.1098/rstb.2013.0623

malized units where the unit of time is 1 msec, the unit of electrical potential is 0.1 V (with an offset of -0.1 for membrane potentials and related terms, such that their normal range stays within the $[0,1]$ normalized bounds), and the unit of current is $1.0x10^{-8}$.

The membrane potential V_m is updated as a function of ionic conductances g with reversal (driving) potentials E as follows:

$$\Delta V_m(t) = \tau \sum_c g_c(t)\overline{g_c}(E_c - V_m(t)) \tag{1.1}$$

with 3 channels (c) corresponding to: e excitatory input; l leak current; and i inhibitory input. Following electrophysiological convention, the overall conductance is decomposed into a time-varying component $g_c(t)$ computed as a function of the dynamic state of the network, and a constant $\overline{g_c}$ that controls the relative influence of the different conductances. The equilibrium potential can be written in a simplified form by setting the excitatory driving potential (E_e) to 1 and the leak and inhibitory driving potentials (E_l and E_i) of 0:

$$V_m^\infty = \frac{g_e\overline{g_e}}{g_e\overline{g_e} + g_l\overline{g_l} + g_i\overline{g_i}} \tag{1.2}$$

which shows that the neuron is computing a balance between excitation and the opposing forces of leak and inhibition. This equilibrium form of the equation can be understood in terms of a Bayesian decision making framework (O'Reilly & Munakata, 2000).

The excitatory net input/conductance $g_e(t)$ or η_j is computed as the proportion of open excitatory channels as a function of sending activations x_i times the weight values w_ij:

$$\eta_j = g_e(t) = \langle x_i w_{ij} \rangle = \frac{1}{n}\sum_i x_i w_{ij} \tag{1.3}$$

The inhibitory conductance is computed via the kWTA function described in the next section, and leak is a constant.

In its discrete spiking mode, Leabra implements exactly the AdEx (adaptive exponential) model (Brette & Gerstner, 2005), which has been found to provide an excellent fit to the actual firing properties of cortical pyramidal neurons (Gerstner & Naud, 2009), while remaining simple and efficient to implement. However, we typically use a rate-code approximation to discrete firing, which produces smoother more deterministic activation dynamics, while capturing the overall firing rate behavior of the discrete spiking model.

We recently discovered that our previous strategy of computing a rate-code graded activation value directly from the membrane potential is problematic, because the mapping between V_m and mean firing rate is not a one-to-one function in the AdEx model. Instead, we have found that a very accurate approximation to the discrete spiking rate can be obtained by comparing the excitatory net input directly with the effective computed amount of net input required to get the neuron firing over threshold (g_e^Θ), where the threshold is indicated by Θ:

$$g_e^\Theta = \frac{g_i\bar{g}_i(E_i - V_m^\Theta) + \bar{g}_l(E_l - V_m^\Theta)}{\bar{g}_e(V_m^\Theta - E_e)} \tag{1.4}$$

$$y_j(t) \propto g_e(t) - g_e^{\Theta} \tag{1.5}$$

where $y_j(t)$ is the firing rate output of the unit.

We continue to use the Noisy X-over-X-plus-1 (NXX1) function, which starts out with a nearly linear function, followed by a saturating nonlinearity:

$$y_j(t) = \frac{1}{\left(1 + \frac{1}{\gamma[g_e(t) - g_e^{\Theta}]_+}\right)} \tag{1.6}$$

where γ is a gain parameter, and $[x]_+$ is a threshold function that returns 0 if $x < 0$ and x if $x > 0$. Note that if it returns 0, we assume $y_j(t) = 0$, to avoid dividing by 0. As it is, this function has a very sharp threshold, which interferes with graded learning learning mechanisms (e.g., gradient descent). To produce a less discontinuous deterministic function with a softer threshold, the function is convolved with a Gaussian noise kernel ($\mu = 0$, $\sigma = .005$), which reflects the intrinsic processing noise of biological neurons:

$$y_j^*(x) = \int_{-\infty}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-z^2/(2\sigma^2)} y_j(z - x) dz \tag{1.7}$$

where x represents the $[g_e(t) - g_e^{\Theta}]_+$ value, and $y_j^*(x)$ is the noise-convolved activation for that value. In the simulation, this function is implemented using a numerical lookup table.

1.2 k-Winners-Take-All Inhibition

Leabra uses a kWTA (k-Winners-Take-All) function to achieve inhibitory competition among units within a layer (area). The kWTA function computes a uniform level of inhibitory current for all units in the layer, such that the $k+1$th most excited unit within a layer is generally below its firing threshold, while the kth is typically above threshold. Activation dynamics similar to those produced by the kWTA function have been shown to result from simulated inhibitory interneurons that project both feedforward and feedback inhibition (O'Reilly & Munakata, 2000). Thus, although the kWTA function is somewhat biologically implausible in its implementation (e.g., requiring global information about activation states and using sorting mechanisms), it provides a computationally effective approximation to biologically plausible inhibitory dynamics.

kWTA is computed via a uniform level of inhibitory current for all units in the layer as follows:

$$g_i = g_{k+1}^{\Theta} + q(g_k^{\Theta} - g_{k+1}^{\Theta}) \tag{1.8}$$

where $0 < q < 1$ (.25 default used here) is a parameter for setting the inhibition between the upper bound of g_k^{Θ} and the lower bound of g_{k+1}^{Θ}. These boundary inhibition values are computed as a function of the level of inhibition necessary to keep a unit right at threshold Θ:

$$g_i^{\Theta} = \frac{g_e^* \bar{g}_e (E_e - \Theta) + g_l \bar{g}_l (E_l - \Theta)}{\Theta - E_i} \tag{1.9}$$

where g_e^* is the excitatory net input without the bias weight contribution — this allows the bias weights to override the kWTA constraint.

In the basic version of the kWTA function, which is relatively rigid about the kWTA constraint and is therefore used for output layers, g_k^Θ and g_{k+1}^Θ are set to the threshold inhibition value for the kth and $k+1$th most excited units, respectively. Thus, the inhibition is placed exactly to allow k units to be above threshold, and the remainder below threshold. For this version, the q parameter is almost always .25, allowing the kth unit to be sufficiently above the inhibitory threshold.

In the *average-based* kWTA version, g_k^Θ is the average g_i^Θ value for the top k most excited units, and g_{k+1}^Θ is the average of g_i^Θ for the remaining $n-k$ units. This version allows for more flexibility in the actual number of units active depending on the nature of the activation distribution in the layer and the value of the q parameter (which is typically .6), and is therefore used for hidden layers.

1.3 Learning rules

The model uses both Leabra learning using the XCAL formulation (see O'Reilly, Wyatte, Herd, Mingus, & Jilk, 2013 Supplemental Information) as well as an extension called LeabraTI on specific projections.

1.3.1 Leabra XCAL

The Leabra XCAL learning rule is based on a contrast between a sender-receiver activation product term (shown initially as just xy – relevant time scales of averaging for this term are elaborated below) and a dynamic plasticity threshold θ_p (also elaborated below), which are integrated in the XCAL learning function:

$$\Delta_{xcal} w_{ij} = f_{xcal}(xy, \theta_p) \tag{1.10}$$

$$f_{xcal}(xy, \theta_p) = \begin{cases} (xy - \theta_p) & \text{if } xy > \theta_p \theta_d \\ -xy(1-\theta_d)/\theta_d & \text{otherwise} \end{cases} \tag{1.11}$$

($\theta_d = .1$ is a constant that determines the point where the function reverses back toward zero within the weight decrease regime – this reversal point occurs at $\theta_p \theta_d$, so that it adapts according to the dynamic θ_p value).

Leabra learning consists of a combination of error-driven and self-organizing factors (O'Reilly & Munakata, 2000; O'Reilly, Munakata, et al., 2013). In the Leabra XCAL formulation, these two factors emerge out of a single learning rule based on a contrast between a sender-receiver activation product term at three time scales:

- **s** = short time scale, reflecting the most recent state of neural activity (e.g., past 100-200 msec). This is considered the "plus phase" – it represents the *outcome* information on the current trial, and in general should be more correct than the medium time scale.

- **m** = medium time scale, which integrates over an entire psychological "trial" of roughly a second or so – this value contains a mixture of the "minus phase" and the "plus phase", but in contrasting it with the short value, it plays the role of the minus phase value, or expectation about what the system thought should have happened on the current trial.

- **l** = long time scale, which integrates over hours to days of processing – this is a threshold term similar to that used in the Bienenstock, Cooper & Munro (BCM) algorithm (Bienenstock, Cooper, & Munro, 1982).

The error-driven aspect of XCAL learning is driven essentially by the following term:

$$\Delta_{xcal-err} w_{ij} = f_{xcal}(x_s y_s, x_m y_m) \tag{1.12}$$

However, consider the case where either of the short term values (x_s or y_s) is 0, while both of the medium-term values are > 0 – from an error-driven learning perspective, this should result in a significant weight decrease, but because the XCAL function goes back to 0 when the input drive term is 0, the result is no weight change at all. To remedy this situation, we assume that the short-term value actually retains a small trace of the medium-term value:

$$\Delta_{xcal-err} w_{ij} = f_{xcal}(\kappa x_s y_s + (1-\kappa) x_m y_m, x_m y_m) \tag{1.13}$$

(where $\kappa = .9$, such that only .1 of the medium-term averages are incorporated into the effective short-term average).

The self-organizing aspect of XCAL is driven by comparing this same synaptic drive term to a longer-term average, as in the BCM algorithm:

$$\Delta_{xcal-so} w_{ij} = f_{xcal}(\kappa x_s y_s + (1-\kappa) x_m y_m, \gamma_l y_l) \tag{1.14}$$

where $\gamma_l = 3$ is a constant that scales the long-term average threshold term (due to sparse activation levels, these long-term averages tend to be rather low, so the larger gain multiplier is necessary to make this term relevant whenever the units actually are active and adapting their weights).

Combining both of these forms of learning in the full XCAL learning rule amounts to computing an aggregate θ_p threshold that reflects a combination of both the self-organizing long-term average, and the medium-term minus-phase like average:

$$\Delta_{xcal} w_{ij} = f_{xcal}(\kappa x_s y_s + (1-\kappa) x_m y_m, \lambda \gamma y_l + (1-\lambda) x_m y_m) \tag{1.15}$$

where $\lambda = .01$ is a weighting factor determining the mixture of self-organizing and error-driven learning influences (as was the case with standard Leabra, the balance of error-driven and self-organizing is heavily weighted toward error driven, because error-gradients are often quite weak in comparison with local statistical information that the self-organizing system encodes).

The weight changes are subject to a soft-weight bounding to keep within the $0-1$ range:

$$\Delta_{sb} w_{ij} = [\Delta_{xcal}]_+ (1 - w_{ij}) + [\Delta_{xcal}]_- w_{ij} \tag{1.16}$$

where the $[]_+$ and $[]_-$ operators extract positive values or negative-values (respectively), otherwise 0.

Finally, as in the original Leabra model, the weights are subject to contrast enhancement, which magnifies the stronger weights and shrinks the smaller ones in a parametric, continuous fashion. This contrast enhancement is achieved by passing the linear weight values computed by the learning rule through a sigmoidal nonlinearity of the following form:

$$\hat{w}_{ij} = \frac{1}{1+\left(\frac{w_{ij}}{\theta(1-w_{ij})}\right)^{-\gamma}} \tag{1.17}$$

where $\hat{w}_{ij}$ is the contrast-enhanced weight value, and the sigmoidal function is parameterized by an offset θ and a gain γ (standard defaults of 1 and 6, respectively, used here).

1.3.2 LeabraTI

LeabraTI extends standard Leabra learning by interleaving its minus and plus phases over temporally contiguous input sequences. In standard Leabra, the minus phase depends on clamped inputs from the sensory periphery to drive the expectation while the plus phase uses clamped outputs from other neural systems to drive the outcome. In LeabraTI, the minus phase expectation is not driven by the sensory periphery, but instead by lagged context represented by deep (Layer 6) neurons. During the plus phase, driving potential shifts back to the sensory periphery. Deep neurons' context is also updated after each plus phase.

LeabraTI was only used to update the synaptic weights between superficial and deep neurons. Inter-areal feedforward and feedback projections bifurcate from the local column, directly synapsing disparate populations of superficial neurons and thus weight updates in these cases were handled by Leabra XCAL equations. In computing the weight update, the standard Leabra delta rule (O'Reilly, 1996) uses the difference in rate between the plus and minus phases of receiving units (y) in proportion to the rate of sending units (x) in the minus phase:

$$\Delta_{leabra} w_{ij} = x^{-}(y^{+} - y^{-})$$

In the LeabraTI framework, deep neurons are considered to be the receiving units as they are the terminus of the descending columnar synapses. However, deep units are proposed to only be active during the minus phase when they drive the prediction, and thus cannot be used to compute an error signal. To address this issue, we invert the LeabraTI delta rule:

$$\begin{aligned}\Delta_{leabrati} w_{ij} &= super^{-}(deep^{+} - deep^{-}) \\ &\approx deep^{-}(super^{+} - super^{-})\end{aligned}$$

Additionally, the temporally extended nature of the algorithm requires that the receiving units represent the current state (time t) and sending units the previous moment's state (time t - 1). While conceptualized as the previous equation, the actual implementation is as follows:

$$\Delta_{leabrati} w_{ij} = super^{+}_{t-1}(super^{+}_{t} - super^{-}_{t})$$

This formulation allows the driving potential of deep neurons to be computed just once using the previous plus phase state of superficial neurons (multiplied by the superficial $\rightarrow$ deep learned weights) and held constant as an input to superficial neurons during the minus phase. This is a gross simplification of the actual biological process of deep neurons, but is vastly more computationally efficient than explicit modeling by computing an additional rate for each deep neuron at each time step. This formulation is also equivalent to the simple recurrent network (SRN) (Elman, 1990; Servan-Schreiber, Cleeremans, & McClelland, 1991), thus providing a potential biological substrate for its computational function.

One limitation of LeabraTI's interleaving of minus and plus phases over time is that the initial minus phase in an input sequence does not have access to the previous moment's context. Even if there was lagged context available, it would represent information from a prior, possibly unrelated input sequence. To address this, weight updates are disabled for the first minus-plus phase pair, and enabled thereafter. In the brain, this process might be facilitated by a neural mechanism that is sensitive to the repetition of inputs over time (e.g., acetylcholine) (Thiel, Henson, Morris, Friston, & Dolan, 2001; Thiel, Henson, & Dolan, 2002).

2 Training Details

A finite state grammar was created in order to probabilistically generate coffee- and tea-making sequences, following the sequences used in Botvinick and Plaut (2004). Coffee sequences consisted of adding coffee grounds to hot water, adding sugar (from a bowl or a packet) and cream–in either order–and then drinking the coffee. Tea-making involves dipping the teabag in the hot water, adding sugar (from either source), and drinking the tea. Tables 1 and 2 define the states of the grammar, and which tasks and subtasks the steps belong to. After specifying the starting world state in the first row, the next six rows specify the unique coffee- and tea-making sequences described above, which are randomly selected from during training. Then the subtasks of pouring the grounds ("pour_ground" and "pg_" steps) and stirring the grounds ("stir_grounds" and "sg_" steps) are defined. Table 1 goes on to define the teabag (tb) and pour sugar packet (ps) sequential actions. Table 2 defines the stir sugar packet (stsp) action that completes the sugar packet method of adding sugar, and then defines the sugar bowl method (open sugar lid (ol), scoop sugar (ss), and stir sugar bowl (stsb). The final actions defined are pouring cream (pc), stirring cream (sc), and finally drinking the beverage (dr). The model and full training/testing program are available online: `http://www.kachergis.com/downloads/coffee_tea_ti.proj`.

Using 2 hidden layers with 24 units each, 100 networks were trained over 200 50-step epochs. The learning rate was 0.5 until epoch 50, dropping to 0.2 until epoch 100, then dropped to 0.1 until epoch 150, and finally dropped to 0.05 for the final 50 epochs. Training accuracy for the final 50 epochs was 94% for the model with 2 24-unit hidden layers, and 95% for the single 48-unit hidden layer model. Normalized error was .016 for the two layer model, compared to .017 for the single layer model.

Table 1: Definition of states in the coffee- and tea-making finite state grammar.

State Name	Visual	Manual	Action	World State
start				cup=cup, 1_handle, clear_liquid; coffee_packet=packet, foil, untorn; sugar_packet=packet, paper, untorn; spoon=spoon; cream_carton=carton, closed; sugar_bowl=cup, lid, 2_handle;
coffee_sugpack_cream				
coffee_sugbowl_cream				
coffee_cream_sugpack				
coffee_cream_sugbowl				
tea_sugpack				
tea_sugbowl				
grounds				
pour_grounds				
pg_fixate_packet	cup	nothing	fixate_coffee_packet	
pg_pick_up_packet	coffee_packet	nothing	pick_up	
pg_pull_open_packet	coffee_packet	coffee_packet	pull_open	coffee_packet= -untorn, +torn
pg_fixate_cup	coffee_packet	coffee_packet	fixate_cup	
pg_pour_packet	cup	coffee_packet	pour	cup=-clear_liquid, +brown_liquid
stir_grounds				
sg_fixate_spoon	cup	coffee_packet	fixate_spoon	
sg_put_down_packet	spoon	coffee_packet	put_down	
sg_pick_up_spoon	spoon	nothing	pick_up	
sg_fixate_cup	spoon	spoon	fixate_cup	
sg_stir	cup	spoon	stir	
teabag				
tb_fixate_teabag	cup	nothing	fixate_teabag	
tb_pick_up_teabag	teabag	nothing	pick_up	
tb_fixate_cup	teabag	teabag	fixate_cup	
tb_dip_packet	cup	teabag	dip	cup=-clear_liquid, +brown_liquid
sugar_packet				
pour_sugar_packet				
ps_fixate_packet	cup	spoon	fixate_sugar_packet	
ps_put_down_spoon	sugar_packet	spoon	put_down	
ps_pick_up_packet	sugar_packet	nothing	pick_up	
ps_pull_open_packet	sugar_packet	sugar_packet	pull_open	sugar_packet=-untorn, +torn
ps_fixate_cup	sugar_packet	sugar_packet	fixate_cup	
ps_pour_packet	cup	sugar_packet	pour	

Table 2: Definition of states in the coffee- and tea-making finite state grammar, part 2.

State Name	**Visual**	**Manual**	**Action**	**World State**
stir_sugar_packet				
stsp_fixate_spoon	cup	sugar_packet	fixate_spoon	
stsp_put_down_packet	spoon	sugar_packet	put_down	
stsp_pick_up_spoon	spoon	nothing	pick_up	
stsp_fixate_cup	spoon	spoon	fixate_cup	
stsp_stir	cup	spoon	stir	
sugar_bowl				
open_sugar_lid				
ol_fixate_sugar	cup	spoon	fixate_sugar_bowl	
ol_put_down_spoon	sugar_bowl	spoon	put_down	
ol_pull_off_lid	sugar_bowl	nothing	pull_off	sugar_bowl= -lid+sugar
scoop_sugar				
ss_fixate_spoon	sugar_bowl	lid	fixate_spoon	
ss_put_down_lid	spoon	lid	put_down	
ss_pick_up_spoon	spoon	nothing	pick_up	
ss_fixate_sugar	spoon	spoon	fixate_sugar_bowl	
ss_scoop_sugar	sugar_bowl	spoon	scoop	spoon=+sugar
stir_sugar_bowl				
stsb_fixate_cup	sugar_bowl	spoon	fixate_cup	
stsb_pour	cup	spoon	pour	spoon=-sugar
stsb_stir	cup	spoon	stir	
cream				
pour_cream				
pc_fixate_carton	cup	spoon	fixate_carton	
pc_put_down_spoon	cream_carton	spoon	put_down	
pc_pick_up_carton	cream_carton	nothing	pick_up	
pc_peel_open	cream_carton	cream_carton	peel_open	cream_carton= -closed+open
pc_fixate_cup	cream_carton	cream_carton	fixate_cup	
pc_pour_cream	cup	cream_carton	pour	cup=+light
stir_cream				
sc_fixate_spoon	cup	cream_carton	fixate_spoon	
sc_put_down_carton	spoon	cream_carton	put_down	
sc_pick_up_spoon	spoon	nothing	pick_up	
sc_fixate_cup	spoon	spoon	fixate_cup	
sc_stir	cup	spoon	stir	
drink				
dr_put_down_spoon	cup	spoon	put_down	
dr_pick_up_cup	cup	nothing	pick_up	
dr_sip	cup	cup	sip	
dr_sip2	cup	cup	sip	cup=-brown_liquid -light+empty
dr_done	cup	cup	say_done	

3 Test Results

Shown in Figure 1, the normal network weights perform very well for almost all test trials, whereas the Non-TI version performs very poorly for nearly all trials. Lesioning the second hidden layer (Hid2 Lesion) impacted performance somewhat, and more for some steps than others. Weakening the context projections had a larger impact on performance, but still resulted in much better performance than the Non-TI network.

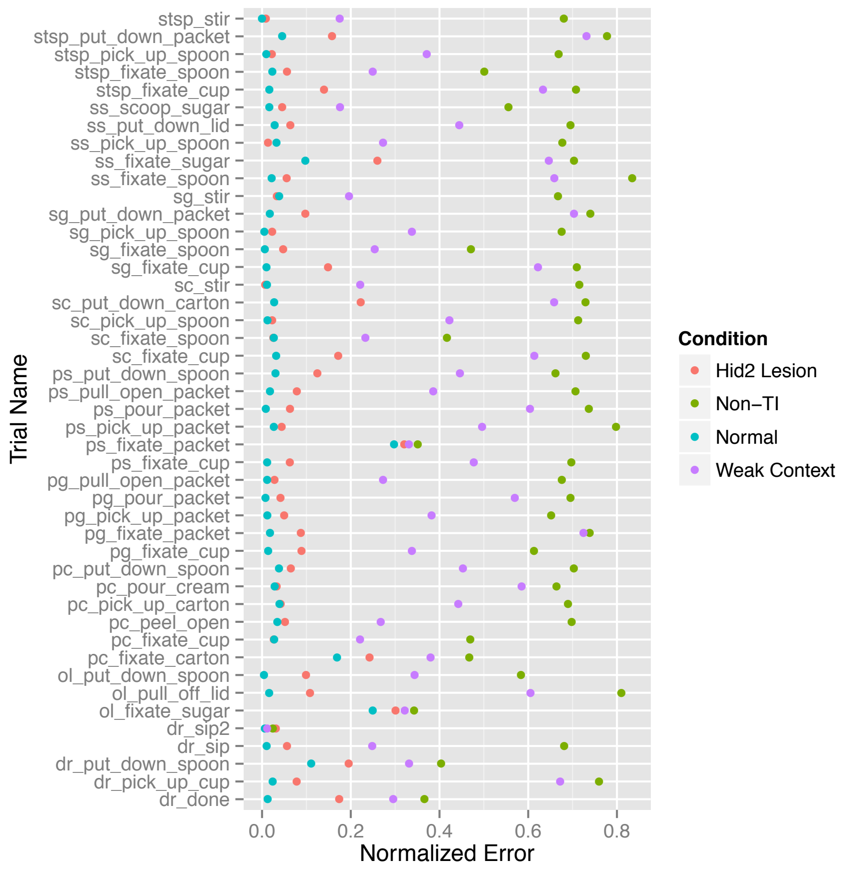

Figure 1: Normalized error for each test trial is shown for different manipulations of 100 trained networks. The Normal (2 hidden layers with 24 units) network structure performs the best, but shows some error for the ol_fixate_sugar and ps_fixate_packet trials. The Non-TI version performs the worst for every trial type. Even the Weak Context performs somewhat better (with great variability–some steps are near-normal, while others are close to Non-TI error rates). Finally, lesioning the second hidden layer (Hid2 Lesion) only moderately decreases performance.

References

Bienenstock, E. L., Cooper, L. N., & Munro, P. W. (1982). Theory for the development of neuron selectivity: Orientation specificity and binocular interaction in visual cortex. *The Journal of Neuroscience*, *2*(2), 32-48.

Botvinick, M., & Plaut, D. C. (2004). Doing without schema hierarchies: A recurrent connectionist approach to routine sequential action and its pathologies. *Psychological Review*, *111*, 395–429.

Brette, R., & Gerstner, W. (2005). Adaptive exponential integrate-and-fire model as an effective description of neuronal activity. *Journal of Neurophysiology*, *94*(5), 3637-3642.

Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, *14*(2), 179–211.

Gerstner, W., & Naud, R. (2009). How good are neuron models? *Science*, *326*(5951), 379-380.

O'Reilly, R. C. (1996). Biologically Plausible Error-driven Learning using Local Activation Differences: The Generalized Recirculation Algorithm. *Neural Computation*, *8*(5), 895-938.

O'Reilly, R. C. (2001). Generalization in interactive networks: The benefits of inhibitory competition and Hebbian learning. *Neural Computation*, *13*(6), 1199-1242.

O'Reilly, R. C., & Munakata, Y. (2000). *Computational explorations in cognitive neuroscience: Understanding the mind by simulating the brain*. MIT Press.

O'Reilly, R. C., Munakata, Y., Frank, M. J., Hazy, T. E., & Contributors. (2013). *Computational cognitive neuroscience* (2nd ed.). URL: `http://ccnbook.colorado.edu`: Wiki Book.

O'Reilly, R. C., Wyatte, D., Herd, S., Mingus, B., & Jilk, D. (2013). Recurrent processing during object recognition. *Frontiers in Psychology*, *4*(124).

O'Reilly, R. C., Wyatte, D., Rohrlich, J., & Herd, S. A. (in preparation). Learning through time in the thalamocortical loops.

Servan-Schreiber, D., Cleeremans, A., & McClelland, J. (1991). Graded state machines: The representation of temporal contingencies in simple recurrent networks. *Machine Learning*, *7*(2-3), 161-193.

Thiel, C. M., Henson, R. N., Morris, J. S., Friston, K. J., & Dolan, R. J. (2001). Pharmacological modulation of behavioral and neuronal correlates of repetition priming. *The Journal of Neuroscience*, *21*(17), 6846-6852.

Thiel, C. M., Henson, R. N. A., & Dolan, R. J. (2002). Scopolamine but not lorazepam modulates face repetition priming: A psychopharmacological fMRI study. *Neuropsychopharmacology*, *27*(2), 282-292.

Last updated: June 27, 2014