

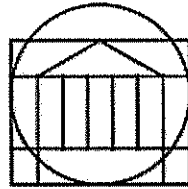
Using a Proxy-Oriented Genetic Algorithm to Find a Millisecond-Scale Model of the Hippocampus

A Dissertation

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia



In Partial Fulfillment

of the Requirements for the Degree

Doctor of Philosophy (Computer Science)

by

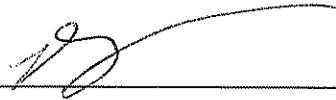
Ashlie Benjamin Hocking

May 2012

© Copyright by
Ashlie Benjamin Hocking
All Rights Reserved
2012

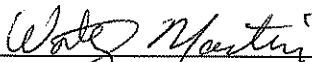
APPROVAL SHEET

This dissertation is submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy (Computer Science)



Ashlie Benjamin Hocking (AUTHOR)

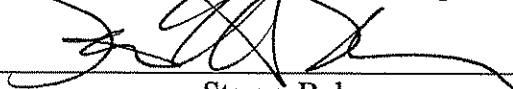
This dissertation has been read and approved by the examining Committee:



Worthy N. Martin (Thesis Advisor)



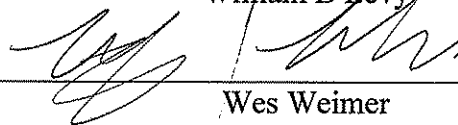
Jim Cohoon (Committee Chairperson)



Steven Boker

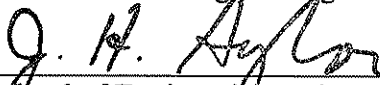


William B. Levy



Wes Weimer

Accepted for the School of Engineering and Applied Science:



Dean, School of Engineering and Applied Science

May 2012

Abstract

The Levy model is a neural network model of the CA3 region of the hippocampus. Previous work with the Levy model has shown success in modeling such hippocampally dependent tasks as trace conditioning, configural learning, spatial navigation, and sequence learning. Learning these tasks require network-scale behavior over simulated time-scales of minutes or longer. Most simulations of the model use simple McCulloch-Pitts neurons operating at time-scales of 15-30 ms.

Replacing the McCulloch-Pitts neurons with Izhikevich neurons allows the model to demonstrate biologically plausible neuron-scale behavior over time-scales of 1 ms and shorter. However, reproducing the network-scale behavior shown using the simpler McCulloch-Pitts neurons becomes more complicated due to the increased number of interacting parameters.

A genetic algorithm is used to explore these interacting parameters. Since the fitness function requires running a simulation of the CA3 region of the hippocampus, a proxy fitness function is used that simulates less than one second of time in the hippocampal model rather than a complete multi-minute simulation. The full fitness function only needs to be evaluated for parameter settings that pass a threshold value for the proxy function. Using a proxy-oriented genetic algorithm, settings were for the extended Levy model so that it can operate at millisecond time scales, demonstrate neuron-scale plausible behavior, while still demonstrating trace conditioning acquisition.

Table of Contents

ABSTRACT	I
TABLE OF CONTENTS.....	II
LIST OF FIGURES.....	IV
LIST OF TABLES	VII
LIST OF SYMBOLS.....	VIII
OVERVIEW	1
NEURAL NETWORKS AND THE LEVY MODEL OF THE HIPPOCAMPUS.....	1
<i>The Levy Model of the Hippocampus.....</i>	3
<i>Demonstrating that the Levy Model Computes Conditional Probabilities</i>	7
<i>The Izhikevich Neuron.....</i>	12
<i>Gamma and Theta Oscillations.....</i>	13
<i>Trace Conditioning.....</i>	14
EXTENDING THE LEVY MODEL TO THE 1 MS TIME-SCALE	18
GENETIC ALGORITHMS	21
METHODOLOGIES	22
MEASURING GAMMA OSCILLATIONS.....	22
MEASURING THETA MODULATED GAMMA OSCILLATIONS.....	23
GENETIC ALGORITHM EXPLORATION	24
<i>Primary Fitness Function (F_1).....</i>	30
<i>Short-Circuit Fitness Function (F_3)</i>	31
<i>Determining the existence of parameter settings for acquiring trace conditioning.....</i>	31

<i>Proxy effectiveness</i>	32
<i>Evaluation efficiency</i>	32
<i>Trajectory efficiency</i>	33
NAÏVE EXPLORATION	33
RESULTS	35
IMPROVEMENT IN BIOLOGICAL PLAUSIBILITY	35
GAMMA OSCILLATIONS	36
THETA MODULATED GAMMA OSCILLATIONS	42
EXISTENCE OF SETTINGS FOR TRACE CONDITIONING ACQUISITION	45
PROXY EFFECTIVENESS	47
EVALUATION EFFICIENCY	51
TRAJECTORY EFFICIENCY	56
CONCLUSIONS	57
CONTRIBUTIONS	57
<i>Extension to Levy Model and Existence Proof of Viable Parameters</i>	57
<i>Effective and Efficient Technique for Discovery</i>	58
FUTURE WORK	60
APPENDIX	67
ALTERNATE RANDOM SEEDS FOR TRACE CONDITIONING EXAMPLE	67
ALL GENERATIONS OF PROXY COMPUTATIONAL SPEED UP	74
SAMPLE TRACE CONDITIONING SCRIPT	75

List of Figures

Figure 1. Feedforward neural network.....	1
Figure 2. Sparsely connected neural network	2
Figure 3. Fully connected neural network.....	2
Figure 4: Number of neurons expected to fire per millisecond over time for different neuron categories and different trials	17
Figure 5. Periodic function used to modulate input	24
Figure 6. Membrane potential of a neuron during the first training trial of trace conditioning.	35
Figure 7. External input. A sequence of 32 patterns of externally activated neurons is presented repeatedly to a simulated neural network.	37
Figure 8. A gamma oscillation appears in the macroscopic activity of the network simulation is obtained	38
Figure 9. Power spectrum for simulations with an input of a repeating sequence (top), with random input (middle), and with no external input (bottom)	39
Figure 10. Power spectrum for simulations with an average firing rate of 2 Hz (top), 4 Hz (middle), and 8 Hz (bottom).....	40
Figure 11. (A) A reordered firing diagram for a simulation run at 2 Hz. (B) A reordered firing diagram for a simulation run at 8 Hz.....	41
Figure 12. Power spectrum of a simulation in the absence of any external input demonstrating emergent gamma oscillations.....	42

Figure 13. Power spectrum in a simulation without external input after training on theta-modulated input.....	43
Figure 14. Power spectrum for simulations in the presence of a theta-modulated input.....	44
Figure 15. Trace conditioning acquisition.....	45
Figure 16. Across trial activity for an individual simulation that demonstrated trace conditioning acquisition.	46
Figure 17. Trace conditioning acquisition for an individual using parameter settings discovered during a genetic algorithm experiment, but with a different random seed than was used during that experiment.....	47
Figure 18. In a genetic algorithm using the short circuit fitness function, false positives as a function of the proxy threshold value, τ	48
Figure 19. In a genetic algorithm using the short circuit fitness function, false negatives as a function of the proxy threshold value, τ	49
Figure 20. In a genetic algorithm using the short circuit fitness function, false negatives (left) and false positives (right) as a function of the proxy value's relative position.....	49
Figure 21. In a genetic algorithm using only the primary fitness function, false positives as a function of the proxy threshold value, τ	50
Figure 22. Speedup of simulation calculating only proxy fitness function over simulation calculating primary fitness function for individuals in generation for genetic algorithms using the short circuit fitness function.....	51

Figure 23. Speedup of simulation calculating only proxy fitness function over simulation calculating primary fitness function for individuals in generation for genetic algorithms using only the primary fitness function.	53
Figure 24. In a genetic algorithm using the short circuit fitness function, the fraction of simulations that did not require evaluating the primary fitness function	54
Figure 25. In a genetic algorithm using only the primary fitness function, the fraction of simulations that did not require evaluating the primary fitness function	55
Figure 26. In a genetic algorithm using the short circuit fitness function, the fraction of simulations that did not require evaluating the primary fitness function	56

List of Tables

Table 1. Properties of the Levy minimal model of the CA3 region of the hippocampus.....	4
Table 2. Typical formulae governing the Levy model.....	6
Table 3. Some terms used in describing time in the Levy model.	7
Table 4. Extension of the Levy Model to the 1 ms timescale	20
Table 5. Important behaviors in trace conditioning	26
Table 6. List of evaluating equations for a trace conditioning episode.....	27
Table 7. A summary of parameters describing the genetic algorithm used in this research.....	28
Table 8. Fitness functions and their mathematical description	29

List of Symbols

$ n_t , n_p $	Number of tone (t) or puff (p) neurons
a, b, c, d, g	Izhikevich parameter (real valued)
$A(t_1, t_2)$	Average activity between times t_1 and t_2
a_d	Desired activity fraction
a_{ij}	Axonal delay from neuron i to neuron j (integer)
B_1, B_2, B_3	Behavior
c	Connectivity percent
c_{ij}	Connectivity (binary 0,1)
$D(t_1, t_2)$	Squared deviation from desired activity between times t_1 and t_2
F_1, F_2, F_3	Fitness function
f	Synaptic failure rate
I_{FF}, I_{FB}	Interneuron activation (real valued)
I_j	Somatic current injection (real valued)
K_0, K_{FF}, K_{FB}	Inhibition constants
$k_{collapse}, k_{preblink}$	Constants describing desirability of puff neuron activity during pre-
k_{blink}, k_{bridge}	defined intervals
M_1, M_2, M_3, \dots	Measure
m_e	Number of externally activated neurons
n	Number of neurons
$N(t_1, t_2, s)$	Number of neurons from set s firing between times t_1 and t_2
n_t, n_p, n_A	Set of tone (t), puff (p), or all (A) neurons

$P(t_1, t_2)$	Fraction of activity due to puff neurons between times t_1 and t_2
p_a	Percentage of neurons firing per second
p_e	Percentage of activity due to external activation
t_{hw}	Dendritic filter half width (integer)
t_l	Axonal delay from interneuron to pyramidal neuron (real valued)
u_j	Secondary Izhikevich somatic variable (real valued)
v_j	Somatic voltage (primary Izhikevich somatic variable, real valued)
w_{ij}	Synaptic weight from neuron i to neuron j (real valued)
x_i	External activation of neuron i (binary 0, 1)
y_i	Dendritic input (real valued)
z_i	Axonal output (binary 0,1)
α	NMDA off-rate decay (real valued)
Δt	Time step (real valued)
Δw_{ij}	Change in synaptic weight (real valued)
λ	Inhibitory decay rate (real valued)
μ	Synaptic modification rate (real valued)
σ_d	Desired standard deviation of activity
σ_s	Sample standard deviation of activity
τ_{on}	NMDA on-rate time constant (real valued)
$\Phi()$	Synaptic failure channel (binary 0,1 output)
Ψ_i	The time when neuron i last fired (real valued)

Overview

Neural Networks and the Levy Model of the Hippocampus

The term “artificial neural network” is used to describe a variety of systems of simulated neurons that differ in both topology and detail of simulated neurons. What these systems have in common is multiple simulated neurons with some of the neurons receiving information from others.

Three broad types of topology include feed-forward (Figure 1), fully connected (Figure 2), and sparsely connected (Figure 3). The hippocampus has a sparsely connected topology, so it is used for this research.

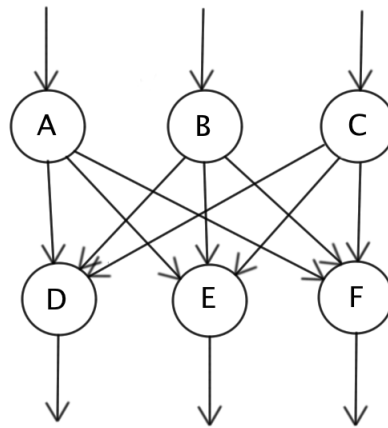


Figure 1. Feedforward neural network. Neurons can be ordered such that signals always travel from top to bottom. No cycles exist.

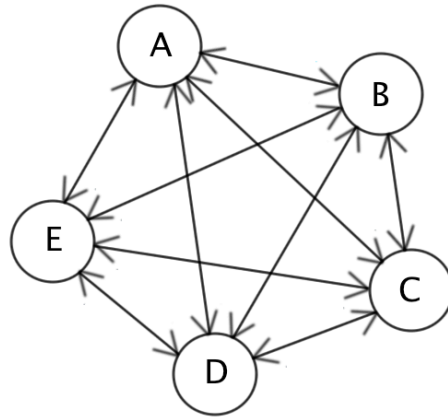


Figure 2. Fully connected neural network. Every neuron is connected to every other neuron. The number of connections necessarily scales as n^2 .

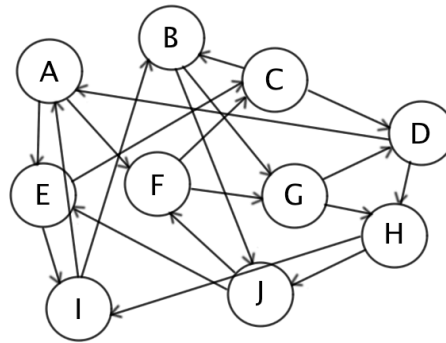


Figure 3. Sparsely connected neural network. Each neuron is connected to multiple other neurons, and cycles will exist. In this example, each neuron is connected to 20% of the neurons in the network. If the relationship of being connected to a fixed percentage of other neurons holds, then the number of connections would scale as n^2 . Alternatively, each neuron might be connected to a fixed number of other neurons, in which case the number of connections scales linearly with n .

Simulated neurons vary from the trivially simple to incredibly complex. Among the simplest simulated neuron is the McCulloch-Pitts neuron. McCulloch-Pitts neurons take input from other neurons, multiply them by a weight and sum those weighted values. The neuron fires only if that sum exceeds a specific threshold. [1] There is no time-scale or memory built in to McCulloch-Pitts neurons, although for topologies analogous to biological neural networks a time-scale can be inferred from the mean activity of the

neurons in the network.

Slightly more complex is a leaky integrate-and-fire neuron. In integrate-and-fire neurons a potential is maintained from one time-step to the next, with the weighted sum of the neuron's inputs for each time-step being added to that potential. If the potential exceeds a threshold, then the neuron fires, and the potential is reset. For leaky integrate-and-fire, the potential is reduced by a fractional amount from one time-step to the next. How much this potential reduces over a time-step can be used to infer a time-scale by comparison to a biological neuron.

Very complex simulated neurons may include details such as chemical diffusion, ion flow, dendritic structure, and more. A moderately complex simulated neuron model is the Hodgkin-Huxley model, which simulates action potentials and their proposed ionic mechanisms. [2] Two simplified versions of this model that capture most of its behavior while requiring fewer computations is the FitzHugh-Nagumo model [3, 4] and the Izhikevich model [5]. The Izhikevich neuron is of special interest as it is “as biologically plausible as the Hodgkin-Huxley model” [5], but requires only about 1% as much processor time (13 FLOPs per neuron per time step versus about 1200 FLOPS per neuron per time step). [6]

The Levy Model of the Hippocampus

The Levy model of the hippocampus is not a neural network model *per se*. It is a statement of hippocampal purpose, a description of hippocampal architecture, a prescription for providing inputs and interpreting outputs (Table 1), and a set of formulae governing how neurons respond to their inputs (Table 2)[1]. More specifically, the model holds that the hippocampus is a sparsely connected sequence learning device that gathers

and randomly mixes, or recodes, inputs from sensory cortices. From the Levy model arise a suite of neural network models that have these features in common.

<ol style="list-style-type: none"> 1. Neurons are thresholded—when inputs (integrated or instantaneous) exceed a threshold, they produce a spike 2. Neurons produce binary output—a spike when threshold is exceeded, and no spike when it threshold has not been exceeded 3. The majority of connections are excitatory 4. Synapses use a Hebbian learning rule to strengthen connections when the pre-synaptic neuron fires shortly before the post-synaptic neuron, or to weaken connections when it does not 5. Recurrent connectivity is sparse and random 6. The majority of neural inputs are recurrent 7. Randomization processes exist 8. Inhibitory interneurons approximate control activity 9. Activity is low, but not too low
--

Table 1. Properties of the Levy minimal model of the CA3 region of the hippocampus.

Typically the Levy model uses McCulloch-Pitts neurons, with the time step of the simulations between 10 and 30 ms. In this scenario, the time step determines both the updating of the internal excitation and the axonal communication lag. For simulations using integrate-and-fire neurons (or Izhikevich neurons, to be discussed later), the time step of the simulations can be taken down below 10 ms, and axonal lag can be modeled more explicitly.

Although the neural network model primarily models the CA3 region of the

hippocampus, the Levy model has the external inputs to the neural network model representing a mixture of the entorhinal cortex (EC) and dentate gyrus (DG) inputs, and the outputs of the model are understood to be interpreted by the CA1 region of the hippocampus. The neural network model primarily models the pyramidal (excitatory) neurons in the CA3 region, along with two interneurons (inhibitory neurons), one for modulating the feed-forward inhibition (i.e., the DC/EG inputs), and one for modulating the feed-back inhibition. Inhibition is implemented via division rather than subtraction, thus inputs are always non-negative.

The topology of the CA3 neural network is modeled as a randomly and sparsely connected network. Each neuron receives inputs from approximately $n \cdot c$ other neurons in the network and similarly provides outputs to approximately $n \cdot c$ other neurons in the network, where n is the number of neurons in the network and c is the connectivity. For networks on the order 100,000 neurons or fewer, c is usually set to 0.1.

Synaptic modification is Hebbian: “neurons that fire together, wire together.” The last equation in Table 1 gives the most common form of synaptic modification, but variants of this rule replace the $z_j(t-1)$ with $\hat{z}_j(t-1)$, where the latter takes into account activity beyond just the last time step [7]. Simulations using the Levy model use synaptic modification during training trials, but not during testing trials. (See also Table 3.)

Synaptic failures are included in the model via the synaptic failure channel (Φ). The synaptic failure channel is a unary function that takes a binary argument (zero or one) and produces a binary output. If the input value is zero, then the output is also zero. However, if the input value is one, then the output value has a probability of f of being zero and a probability of $(1 - f)$ of being one. Thus, the synaptic failure channel can be

considered a Bernoulli random variable (with success probability of $1 - f$) multiplied by its input. Including synaptic failures in the model improves performance for larger networks with lower activity levels. [8]

Another important feature of the Levy model is activity control. The equations in Table 1 give one example of how activity control is maintained through shunting, feedback, and feed-forward inhibition. A simpler method, competitive activity control, fires neurons with the highest somato-dendritic excitation. [9]

Somato-dendritic excitation/inhibition:

$$(Eq. 1) \quad y_j(t) = \frac{\sum_{i \in n_A} w_{ij} c_{ij} \Phi(z_i(t-1))}{\sum_{i \in n_A} w_{ij} c_{ij} \Phi(z_i(t-1)) + K_0 + K_{FF} \sum_{i \in n_A} x_i(t-1) + K_{FB} \sum_{i \in n_A} z_i(t-1)}$$

Output:

$$(Eq. 2) \quad z_j(t) = \begin{cases} 1 & \text{if } y_j(t) \geq 1/2 \vee x_j(t) = 1 \\ 0 & \text{otherwise} \end{cases}$$

Synaptic modification:

$$(Eq. 3) \quad \Delta w_{ij}(t) = \mu z_j(t) (z_i(t-1) - w_{ij}(t-1))$$

Table 2. Typical formulae governing the Levy model. w_{ij} is the weight from neuron i to j ; c_{ij} is a $\{0,1\}$ value defining whether there is a connection from neuron i to j ; Φ is a $\{0, 1\}$ synaptic failure channel (see text for more details); z_i is a $\{0, 1\}$ value based on whether neuron i fires, n_A is the set of all neurons, K_0 is a resting shunting inhibition constant; K_{FF} is a feed-forward inhibition constant; x_i is a $\{0,1\}$ value based on whether neuron i was driven externally; K_{FB} is a feedback inhibition constant; and μ is the synaptic modification rate.[1] Variants of the model might use different rules for synaptic modification [7] or for determining which neurons fire[9].

Term	Definition
Time step (Δt)	The smallest unit of time in a neural network model that does not use continuous time.
Interval	Time frame (composed of multiple time steps) over which a stimulus (or lack of stimulus) is active.
Trial	A trial is composed of several time steps (anywhere from 3 to a thousand).
Training trial	Trial during which a sequence of external neuronal patterns are presented to the simulated hippocampus and synaptic modification occurs.
Testing trial	Trial during which a portion of a training sequence is presented to the simulated hippocampus and measurements are made to determine how well the correct sequence is recalled. Synaptic modification is disabled during testing.
Episode	Multiple training trials and one or more testing trials usually representing an experiment performed on a lab animal.

Table 3. Some terms used in describing time in the Levy model.

Demonstrating that the Levy Model Computes Conditional Probabilities

In understanding how the Levy model provides sequence prediction functionality, I demonstrate that the equations in table 1 can be shown to compute conditional probabilities. In the conditional probability based model for the hippocampus, the CA3 region generates the probability that a particular event will occur in the immediate future, given that a particular sequence of events has just occurred. The firing pattern of the CA3 encodes this probability, based on external input from the entorhinal cortex and dentate

gyrus, as well as recurrent input from within the CA3 itself, which provide information regarding previous events. One method of decoding a CA3 forecast [1] assumes that a CA3 pyramidal neuron that fires when an event occurs also fires as a signal that the same event is expected to occur.

Mathematically, the necessary forecasting calculation for the idealized CA3 neuron is $P(Z_j(t) = 1 | \vec{Z}(t - \Delta t) = \vec{z})$, where $Z_j(t)$ is the contribution of neuron j to determining whether or not an event is expected to occur, and $\vec{Z}(t - \Delta t)$ is recurrent input to the CA3. For this to be a forecasting calculation, the event $\vec{Z}(t - \Delta t) = \vec{z}$ must precede the event $Z_j(t) = 1$. We will use this time ordering of events for the remainder of this discussion, so the implicit term t will be dropped from our equations.

For a single pyramidal neuron in the mammalian CA3, the number of afferent neurons in \vec{Z} is on the order of 10^4 , so even if the individual components of \vec{Z} are constrained to be binary, there are $2^{10,000}$ possible values that \vec{Z} can attain. Therefore, most values of \vec{Z} will never be experienced, so $P(Z_j = 1 | \vec{Z} = \vec{z})$ will not be calculable simply by examining prior history. However, according to Bayes' Theorem,

$$(Eq. 4) \quad P(Z_j = 1 | \vec{Z} = \vec{z}) = \frac{P(\vec{Z} = \vec{z} | Z_j = 1)P(Z_j = 1)}{P(\vec{Z} = \vec{z} | Z_j = 0)P(Z_j = 0) + P(\vec{Z} = \vec{z} | Z_j = 1)P(Z_j = 1)}$$

This can be combined with an assumption of approximate conditional independence:

$$(Eq. 5) \quad P^*(\vec{Z} = \vec{z} | Z_j = 1) = \prod_{i=1}^n P(Z_i = z_i | Z_j = 1)$$

where $P^*(\vec{Z} = \vec{z} | Z_j = 1)$ indicates the approximation that the components of \vec{Z} are independent of one another given the output of neuron j .

The individual probabilities $P(Z_i = z_i | Z_j = 1)$ can be estimated by the observed average value of Z_i , prior to neuron j firing (that is, prior by Δt) since the values it can take on are only zero and one. Specifically,

$$\begin{aligned} P(Z_i = 1 | Z_j = 1) &= \hat{Z}_{ij(1)}, & P(Z_i = 0 | Z_j = 1) &= 1 - \hat{Z}_{ij(1)}, \\ \text{(Eq. 6)} \quad P(Z_i = 1 | Z_j = 0) &= \hat{Z}_{ij(0)}, & \text{and } P(Z_i = 0 | Z_j = 0) &= 1 - \hat{Z}_{ij(0)}, \end{aligned}$$

where

$$\text{(Eq. 7)} \quad \hat{Z}_{ij(1)} \stackrel{\text{def}}{=} E[Z_i | Z_j = 1] \quad \text{and} \quad \hat{Z}_{ij(0)} \stackrel{\text{def}}{=} E[Z_i | Z_j = 0].$$

It is important to note that there is no direct relationship between statistics $\hat{Z}_{ij(1)}$ and $\hat{Z}_{ij(0)}$.

Similarly, $P(Z_j = 1) = \hat{Z}_j$ where $\hat{Z}_j \stackrel{\text{def}}{=} E[Z_j]$. Since the z_i terms are binary, the preceding equations can be combined as:

$$\text{(Eq. 8)} \quad P^*(Z_j = 1 | \vec{Z} = \vec{z}) = \frac{\hat{Z}_j \prod_{i=1}^n Z_{ij(1)}^{z_i} (1 - \hat{Z}_{ij(1)})^{1-z_i}}{\hat{Z}_j \prod_{i=1}^n Z_{ij(1)}^{z_i} (1 - \hat{Z}_{ij(1)})^{1-z_i} + (1 - \hat{Z}_j) \prod_{i=1}^n Z_{ij(0)}^{z_i} (1 - \hat{Z}_{ij(0)})^{1-z_i}}.$$

This equation now similar to Eq. 1, except that it uses products instead of summations.

Logarithmic manipulation changes the products into summations.

Dividing the numerator into the denominator gives:

$$\text{(Eq. 9)} \quad P^*(Z_j = 1 | \vec{Z} = \vec{z}) = \left(1 + \frac{(1 - \hat{Z}_j)}{\hat{Z}_j} \prod_{i=1}^n \frac{Z_{ij(0)}^{z_i} (1 - \hat{Z}_{ij(0)})^{1-z_i}}{Z_{ij(1)}^{z_i} (1 - \hat{Z}_{ij(1)})^{1-z_i}} \right)^{-1}.$$

Replacing the probability calculation with an odds calculation yields:

$$\text{(Eq. 10)} \quad \frac{P^*(Z_j = 1 | \vec{Z} = \vec{z})}{P^*(Z_j = 0 | \vec{Z} = \vec{z})} = \frac{\hat{Z}_j}{1 - \hat{Z}_j} \prod_{i=1}^n \left(\frac{1 - \hat{Z}_{ij(1)}}{1 - \hat{Z}_{ij(0)}} \right) \left(\frac{\hat{Z}_{ij(1)}^{z_i} (1 - \hat{Z}_{ij(0)})}{\hat{Z}_{ij(0)}^{z_i} (1 - \hat{Z}_{ij(1)})} \right)^{z_i}$$

To satisfy its role of calculating conditional probability, the neuron fires if the odds exceed some threshold (φ), where $\varphi=1$ is a logical but non-mandatory choice. This requirement can be written as:

$$(Eq. 11) \quad \frac{\hat{Z}_j}{1 - \hat{Z}_j} \prod_{i=1}^n \left(\frac{1 - \hat{Z}_{ij(1)}}{1 - \hat{Z}_{ij(0)}} \right) \left(\frac{\hat{Z}_{ij(1)}(1 - \hat{Z}_{ij(0)})}{\hat{Z}_{ij(0)}(1 - \hat{Z}_{ij(1)})} \right)^{z_j} > \varphi.$$

Taking the natural logarithm of both sides and using the substitutions

$$v_{ij} \stackrel{def}{=} \ln \left(\hat{Z}_{ij(1)}(1 - \hat{Z}_{ij(0)}) / \hat{Z}_{ij(0)}(1 - \hat{Z}_{ij(1)}) \right), K_0 \stackrel{def}{=} \ln \varphi, K_{1(j)} \stackrel{def}{=} \ln \left((1 - \hat{Z}_j) / \hat{Z}_j \right), \text{ and}$$

$$K_{2(j)} \stackrel{def}{=} \sum_{i=1}^n \ln \left((1 - \hat{Z}_{ij(0)}) / (1 - \hat{Z}_{ij(1)}) \right), \text{ simplifies Eq. 11 to:}$$

$$(Eq. 12) \quad \sum_{i=1}^n v_{ij} z_i > K_0 + K_{1(j)} + K_{2(j)},$$

which can trivially be rewritten as:

$$(Eq. 13) \quad 2 \sum_{i=1}^n v_{ij} z_i > \sum_{i=1}^n v_{ij} z_i + K_0 + K_{1(j)} + K_{2(j)}, \text{ or}$$

$$y_j(t) \stackrel{def}{=} \frac{\sum_{i=1}^n v_{ij} z_i(t - \Delta t)}{\sum_{i=1}^n v_{ij} z_i(t - \Delta t) + K_0(t - \Delta t) + K_{1(j)}(t - \Delta t) + K_{2(j)}(t - \Delta t)} > \frac{1}{2},$$

where the explicit dependence on time has been reintroduced for clarity. Equation 13 now closely resembles Eq. 1 and 2.

Therefore, the only information required for a neuron to create reasonable forecasts of the future are the state of its afferent neurons ($z_i(t - \Delta t)$, transmitted through synapses), the expectation of the neuron itself firing (\hat{Z}_j) in a computational interval, the expectation of its afferent neurons firing when the neuron fires ($\hat{Z}_{ij(1)}$), and the

expectation of its afferent neurons firing when the neuron is quiescent . Mechanisms exist for the neuron to generate approximations for each piece of necessary information.

A neuron's post-synaptic excitability encodes the expectation of its own firing. That is, lower expectation of firing leads to lower excitability. This naturally arises out of the role that $K_{I(j)}$ plays in Eq. (13). Specifically, decreasing the value of \hat{Z}_j increases the value of $K_{I(j)}$, which decreases the value of y_j .

The expectation of an afferent neuron recently firing given that this neuron is firing ($\hat{Z}_{ij(1)}$) is the statistical correlate of the post-synaptic modification rule described in Eq. 3 and elsewhere. This post-synaptic modification rule only modifies the synaptic weights when the post-synaptic neuron (Z_j) fires. When the post-synaptic neuron fires, the weight is strengthened if the pre-synaptic (afferent) neuron fired in the preceding interval, and weakened if the pre-synaptic neuron was quiescent. Furthermore, Levy et al. [10], demonstrate that the modification rule:

$$\text{(Eq. 14)} \quad \Delta w_{ij}(t) = \mu z_j(t) \left(z_i(t - \Delta t) - w_{ij}(t) \right)$$

under assumptions of stationarity and ergodicity leads to $w_{ij} \rightarrow \hat{Z}_{ij(1)}$ as $t \rightarrow \infty$.

Similarly, the expectation of an afferent neuron recently firing when this neuron is quiescent ($\hat{Z}_{ij(0)}$) is the equivalent of habituation. The synapse becomes habituated to afferent input if it is not followed by a post-synaptic spike, and this habituation decays in the absence of afferent input [10]. The synaptic modification equation of this habituation-like process is:

$$(Eq. 15) \quad \Delta w_{ij}^-(t) = \mu(1 - z_j(t)) \left(z_i(t - \Delta t) - w_{ij}^-(t) \right)$$

which leads to $w_{ij}^- \rightarrow \hat{Z}_{ij(0)}$ under the same assumptions as those made for Eq. 14.

The Izhikevich Neuron

The Izhikevich neuron model is a variant of the FitzHugh-Nagumo model, altered to match the behavior of many different types of neurons through the adjustment of six parameters, several of which are tightly coupled. The Izhikevich equations are:

$$(Eq. 16) \quad \begin{aligned} \frac{dv}{dt} &= 0.04v^2 + fv + g - u + I, & \frac{du}{dt} &= a(bv - u) \\ \text{if } v &\geq 30 \text{ mV, then} & \begin{cases} v = c \\ u = u + d \end{cases} \end{aligned}$$

In the paper where he introduces his new neuron model [5], he uses $f=5$ and $g=140$, which were obtained by fitting the membrane voltage dynamics to that of a cortical neuron, but when modeling his integrator neuron (which has strong similarities to the pyramidal neuron in the hippocampus), he uses $f=4.1$ and $g=108$. Unless specified otherwise, the latter values are the values used in the current research.

What is most significant about the Izhikevich neuron is that it is as biologically plausible as the Hodgkin-Huxley model, while being approximately as computationally efficient as an integrate-and-fire model. [5] In the Izhikevich equations, v corresponds to the membrane potential of the neuron (which have been recorded for many different types of neurons), and u acts as a membrane recovery variable. This membrane recovery variable does not correspond to a single measurable phenomenon but is intended to represent the activation of potassium ionic currents and the inactivation of sodium ionic currents. [5]

Gamma and Theta Oscillations

With the introduction of representative neuron spiking behavior to the Levy model, the study of gamma and theta oscillations as they relate to that model becomes possible.

Gamma oscillations are cycles in neural activity that occur at intervals of 25-100 Hz [11], and researchers have investigated their connection to consciousness [12, 13].

The magnitude of gamma oscillations are typically much smaller than the magnitude of theta oscillations.

Gamma oscillations are present in a sparsely connected random neural network containing only 1,000 neurons (800 excitatory and 200 inhibitory), when the excitatory neurons mimic cortical neurons. These gamma oscillations are an emergent property of his neural network, in that there is no component driving the neurons with a gamma-like frequency [5]. A research question is whether gamma oscillations are also present in my modified version of the Levy model that uses the Izhikevich integrator neuron.

Theta oscillations in the hippocampus are cycles in neural activity that occur at intervals of 4-10 Hz [14]. They have been observed to occur in rats while active [15] and in humans during REM sleep and from the transition from sleep to waking [16].

Trace Conditioning

In humans, trace conditioning experiments involve sounding a tone, a brief delay, and then a mild electric shock. Human subjects are said to have acquired trace conditioning when a galvanic skin response is detectable *prior* to the delivery of the shock. In rabbits, trace conditioning experiments involve a tone, a brief delay, and then a puff of air to the eye. Rabbits are said to have acquired trace conditioning when they blink immediately *prior* to the air puff. (More precisely, the rabbit closes its nictitating membrane, which is essentially a translucent inner eyelid and serves the same purpose as a blink would for humans.) More generally, trace conditioning involves the pairing of a conditioned stimulus (tone) with an unconditioned stimulus (air puff), resulting in a response (blink), with the two stimuli separated by a trace interval (on the order of 500 ms). Trace conditioning is acquired when a response (blink) begins just before the end of the trace interval (regardless of the occurrence of the unconditioned stimulus). Technically, the “blink” due to the unconditioned stimulus is known as an unconditioned response and the blink due to the conditioned stimulus is known as a conditioned response.

In neural network computer simulations, trace conditioning is modeled by identifying a subset of *all* hippocampally modeled neurons (n_A) to be *tone neurons* (n_t) and another subset to be *puff neurons* (n_p) (approximately 2-5% of all neurons make up each subset). During the *tone interval* (the first 150 ms of a trial), the tone neurons are quasi-randomly activated such that their activation accounts for 20-50% of the overall network activity ($a_d|n_A|$). Thus, during the tone interval the tone neurons fire at about 10 times the average firing rate. Similarly, during the *puff interval* (the last 100 ms of each trial), the puff neurons fire at about 10 times the average firing rate. After 200 training

trials, a testing trial is initiated during which only the tone neurons are activated externally (and only during the tone interval), and the resultant activity is recorded to determine the fraction of puff neurons that are activated at relevant time intervals as defined in Table 3. The normal functioning of the network involves synaptic modification; however, synaptic modification does not occur during the testing trial.

Among the hippocampally dependent tasks the Levy model has been successful with, trace conditioning is the only one with a time component to its measure of success. [7] However, although the original Levy model (Table 1) has been used at timescales below 10 ms, it has demonstrated success with trace conditioning only at scales longer than 20 ms [1].

An episode exhibits the *target behavior* (B_I) when the activity of the puff neurons during the *blink interval* of a test trial is more than one third the external activation percentage of the puff neurons during the *puff interval* of a training trial. For example, in Figure 1c the activity of the puff neurons averaged over the blink is more than half the external activation percentage during the puff interval. Two forms of near-target behavior are known as a *bridge*, when the puff neurons are activated at a reasonably high level, but not until the puff interval; and a *collapse*, when the puff neurons are activated at a reasonably high level, but too early. Poor performance is when they are never sufficiently activated.

In addition to the number of puff neurons that are activated during particular intervals, it is also instructive to examine the total number of neurons that fire at each time step. The average number of times all neurons fire per simulated second is measured in Hertz (Hz), with the desired value varying from 0.5 Hz to 2.5 Hz. The instantaneous

firing rate is found by taking the number of neurons firing in a time-step divided by the total number of neurons times the number of time-steps in a second. Good activity (B_3) is activity that stays close to the desired activity rate. A summary of these behaviors can be found in Table 5.

Several factors determine the computational requirements of an episode, but the primary factors are the number of neurons in the model ($|n_A|$), the fraction of neurons firing in a simulated second (p_a), the number of simulated seconds in a trial (t_i), the number of trials in an episode (m_i), and the connectivity of the network (c). Of these factors, t_i and m_i are defined by the task being studied (trace conditioning here). The remaining factors are constrained by the biology being simulated. For most animals, the local connectivity of the hippocampus is approximately 10%, and the average firing rate for each neuron is about 0.5 to 2.5 Hz, yielding $c \approx 0.1$ and $p_a \approx 50$ to 250%.

The number of neurons, $|n_A|$, varies significantly from one species to another. In mammals typically used in hippocampal studies, the number of neurons in the CA3 region of the hippocampus varies from about 150 to 220 thousand in the mouse [17] to about 2 to 3 million for humans [18], suggesting a lower bound for $|n_A|$ of 1.5×10^5 . Most calculations in the model depend linearly on the number of active neurons ($|n_A|p_at_im_i$) in an episode, with the dominant computational requirements being from those calculations that depend linearly on the number of active synapses ($|n_A|^2cp_at_im_i$), where the additional factor of $|n_A|c$ defines the number of synapse per neuron. Thus, simulating the entire CA3 region of the hippocampus is presently infeasible when running thousands of episodes.

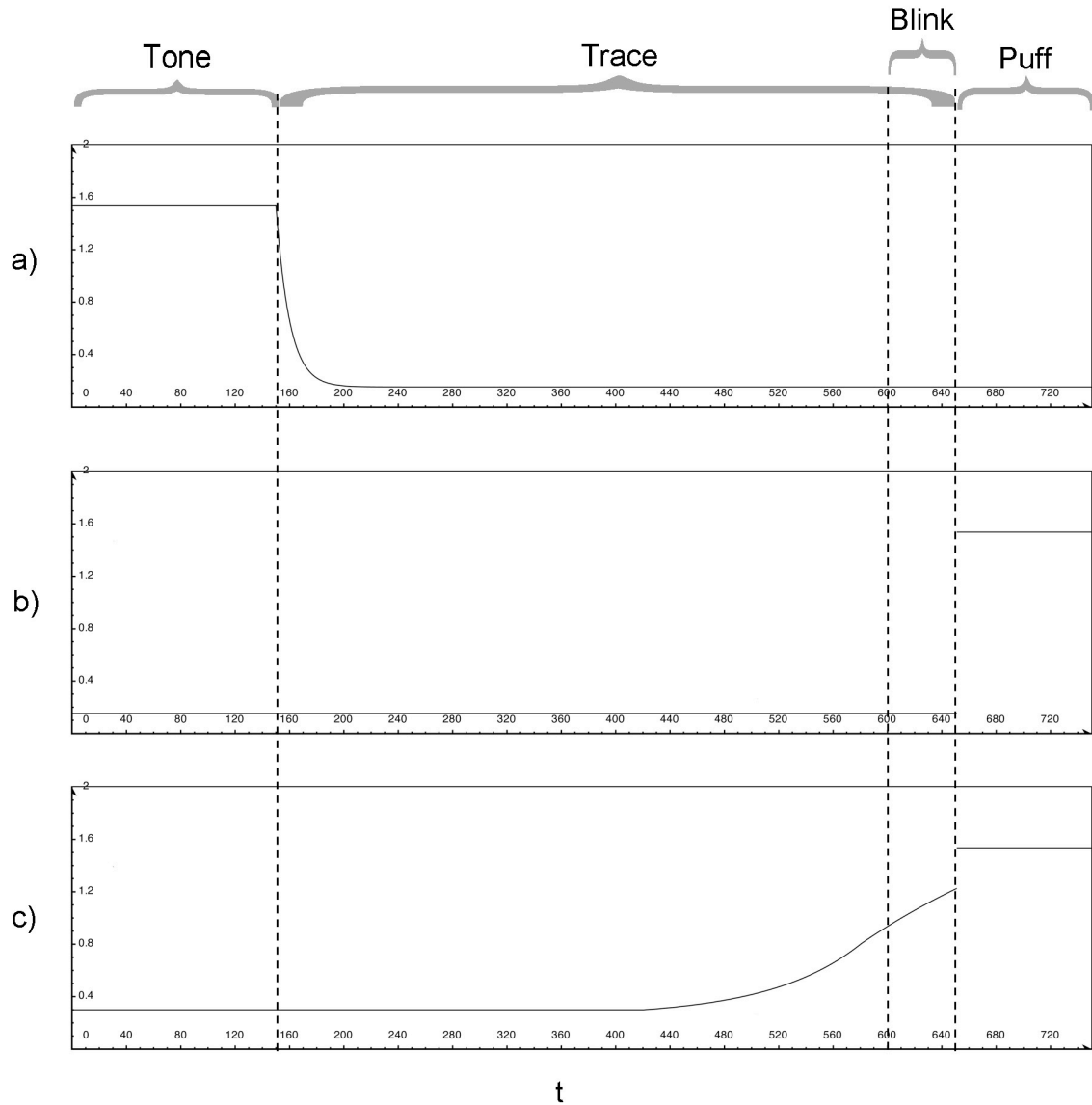


Figure 2: Number of neurons expected to fire per millisecond over time for different neuron categories and different trials. The top graph (a) represents the number of tone neurons expected to fire on the first training trial. The middle graph (b) represents the number of puff neurons expected to fire on the first training trial. The last graph (c) represents the number of puff neurons expected to fire on the last training trial after trace conditioning has been acquired.

The neural network model being studied here has 2,048 neurons (using fewer neurons results in too little recurrent activity) with each neuron connected to 205 other neurons ($c \approx 0.1$). Of the 2,048 neurons, between 41 and 102 (depending on the value of

p_e – see Table 6) are selected as the tone neurons and a disjoint set of neurons of the same size are selected as the puff neurons.

Extending the Levy Model to the 1 ms Time-Scale

Several attempts have been made to extend the Levy model below 15 ms. These extensions begin to break down, however, as the time scale approaches 1 ms [16D 19]. The first problem one sees in extending the model to shorter time scales is that to capture the biological reality of neurons firing approximately only every 500 ms (on average), there are too few neurons firing in a single time-step to provide reliable inputs for the next time-step. For example, with 2,000 neurons with an average firing rate of 2.5 Hz in a simulation running at 1 ms, only 5 neurons are firing per time-step *on average*. With each neuron receiving input from only 10% of the other neurons, there are many times when no neuron would receive input from more than one other neuron, but if inhibition is lowered enough to allow a single input to cause another neuron to fire, there would be insufficient inhibition to prevent run away activity.

Early approaches to solve this problem relied upon leaky integrate-and-fire neurons so neurons would be able to integrate input over several time-steps. This approach worked in simulations requiring no concrete representation of time, but in simulations where time was an important component such as with trace conditioning (discussed later), time was not adequately modeled. Specifically, the learnable trace interval in the trace conditioning task did not agree with experimental results.

As a means of addressing this issue, I replaced the leaky integrate-and-fire neuron used in previous attempts with a neuronal model created by Izhikevich. (See

<http://github.com/BenHocking/NeuroJet> for the source code for the neural network.) The

Izhikevich neuron is “as biologically plausible as the Hodgkin-Huxley model”[5], but requires only about 1% as much processor time (13 FLOPs per neuron per time step versus about 1200 FLOPS per neuron per time step[6]). Adding the Izhikevich neuron requires six (6) new parameters. However, the Izhikevich model of the neuron is really more a model of the soma, as it provides no details about synaptic, axonal, or dendritic functionality. The additional synaptic details include the NMDA-R on-rate and off-rate time constants (2 new parameters). The additional axonal details include the axonal delay from pyramidal neuron (excitatory) to interneuron (inhibitory) as well as the lower and upper bounds for the axonal delay from pyramidal to pyramidal neuron (3 new parameters). A dendritic filter was added which could require a large number of free parameters, but in practice only added one (1) new parameter: dendritic filter width. Furthermore, the interneuron model was updated, but not to use Izhikevich neurons, but rather with two (2) new parameters for excitation decay and for pyramidal-interneuron synaptic modification. All combined this resulted in 14 new parameters (while one parameter, α , is deprecated). *Discovering viable settings for these parameters has demonstrated that the Levy model with these extensions is sufficient to operate neurophysiologically at timescales as short as 1 ms* [20, 21]. The new equations are shown in Table 4.

Synaptic current injection:	$s_j(t) = \sum_i w_{ij} c_{ij} \Phi(z_i(t - a_{ij}))$
Dendritic current injection:	$I_j(t) = \sum_{t_d=1}^{t_{hw}} 1 - e^{-t_{hw} t_d / 5} s_j(t - t_d) + \sum_{t_d=t_{hw}+1}^2 t_{hw} e^{-t_{hw}(t_d - t_{hw})/5} s_j(t - t_d)$
Somatic current injection:	$y_j(t) = \frac{I_j(t)}{I_j(t) + K_0 + K_{FF} I_{FF} + K_{FB} I_{FB}}$
Somatic differential equations:	$\frac{dv_j}{dt} = 0.04v_j^2(t) + 4.1v_j(t) + 108 - u_j(t) + gy_j(t); \quad \frac{du_j}{dt} = a(bv_j(t) - u_j(t))$
Somatic firing:	$v_j(t) = c; u_j(t) = u_j(t - \Delta t) + d; z_j(t) = 1 \quad \text{if } v_j(t) > 30 \vee x_j(t) = 1$ $z_j(t) = 0 \quad \text{otherwise}$
Interneuron activation:	$I_{FF}(t) = \lambda I_{FF}(t - \Delta t) + (1 - \lambda) \sum_{i \in n_A} x_i(t - t_i)$ $I_{FB}(t) = \lambda I_{FB}(t - \Delta t) + (1 - \lambda) \sum_{i \in n_A} z_i(t - t_i)$
Synaptic modification:	$\Delta w_{ij}(t) = \mu z_j(t) (\hat{z}_j(t - \Delta t) - w_{ij}(t))$
Pre-synaptic time-averaging:	$\hat{z}_j(t) = \begin{cases} t - \Psi_i / \tau_{on} & \text{if } \tau_i \leq \tau_{on} \\ \alpha \hat{z}_j(t - \Delta t) & \text{otherwise} \end{cases}$

Table 4. Extension of the Levy Model to the 1 ms timescale. Parameters not discussed in Table 1 are the dendritic filter half-width (t_{hw}), the Izhikevich parameters (a-d, g), interneuron decay rate (λ), axonal delays (a_{ij} , t_i), on-rate time constant (τ_{on}), off-rate decay rate (α), and Ψ_i , the time when neuron i last fired. For summations, the index i is over all neurons, and t is over time-steps.

Genetic Algorithms

Genetic algorithms (GAs) are a form of evolutionary algorithms that are modeled after biological evolutionary genetic processes. They include features such as selection of the fittest individuals, reproduction, cross-over, and mutation. GAs are a form of heuristic search that do not require that the objective function have a known derivative (unlike gradient descent) and that can have multiple permutations evaluated in parallel (unlike traditional implementations of simulated annealing). The objective function in genetic algorithms is referred to as its fitness function. In the work done here, the fitness function will always be defined such that maximal values of the fitness function are considered optimal.

Methodologies

Measuring gamma oscillations

Simulations designed to analyze gamma and theta oscillations used a time step resolution of 3.575 ms, between 4,000 and 6,000 neurons, and adjusted inhibition parameters so that average activity was approximately 2 Hz.

A fast Fourier transform is used to generate a power spectrum of the total network activity, $\sum_{i=1}^n z_i(t)$, with a brick wall filter due to the resolution, or time-step size, of the simulation. For example, if the time step size is 3.575 ms, the brick wall filter is, as a result of this time step size, approximately 280 Hz. Results for the power spectrum are therefore only shown up to 140 Hz. The result of the fast Fourier transform is multiplied by its complex conjugate and divided by the sample size to achieve the power spectrum across frequencies. The resulting power spectrum is then collected into bins of width 0.1 Hz. Results of the fast Fourier transform are shown in linear scale, and are also shown in logarithmic scale as the gamma oscillation becomes less prominent.

The first experiment performed to measure gamma oscillations was to train the neural network on a simple repeating sequence of 32 patterns of externally activated neurons (see Fig. 5 for details). Each pattern was presented to the network for 143 ms, so that one sequence of 32 patterns required a total simulated time of 4.576 seconds. After the network learns the sequence, synaptic modification is turned off, and the network is driven for another 500 trials, during which the activity of the network during each time-step (3.575 ms) is measured.

The second and third experiments to perform to measure gamma oscillations use the trained neural network from the first experiment. In the second experiment, the repeating sequence is replaced with random noise, and in the third experiment no external input is provided, so that all activity is recurrent.

In the fourth set of experiments performed to measure gamma oscillations the strength of the divisive inhibition is reduced so that the network will have a higher average firing rate. Whereas the first three experiments all have an average firing rate of 2 Hz, in the fourth set of experiments 4 and 8 Hz scenarios are examined.

Measuring theta modulated gamma oscillations

Theta-modulated input was presented using several different techniques. In all cases, the input was modulated by the changing the probability a neuron would be externally activated. The simplest technique involved using a sine wave such that the probability a neuron would be externally activated is

$$\text{(Eq. 17)} \quad p_e = A + B \sin(\omega_\theta t),$$

where $\omega_\theta \approx 43.91$ Hz corresponding with a theta frequency of approximately 7 Hz. B was set to one-half of A such that the maximum probability of being externally activated was three times the minimum probability. To achieve some asymmetry in the theta oscillations, a second periodic function, shown in Fig. 1, was also used, although only for the last simulation. This periodic function, $S(x)$, is given by

$$\text{(Eq. 18)} \quad S = \begin{cases} 0, & 5\pi/3 < x < 2\pi \\ -1 + x^{0.4} e^{x/1.8}, & 0 < x < 5\pi/3 \end{cases}$$

As with the sine wave, simulations using this periodic function would have neurons

externally activated with a probability

$$(Eq. 19) \quad p_e = A + B \cdot S(\omega_\theta t - \pi/3)$$

The frequency at which the input was modulated was fixed for some simulations and allowed to randomly vary (from cycle to cycle) in other simulations. The period of modulation would be 143 ms when fixed and between 133 and 153 ms when allowed to vary.

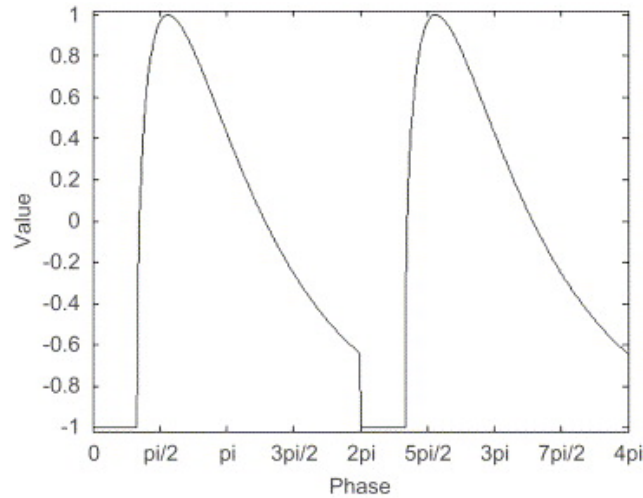


Figure 3. Periodic function used to modulate input. This periodic function (Eq. (18)) is designed to crudely approximate the reaction of neurons in the presence of changing stimuli.

Genetic algorithm exploration

The basic structure of the genetic algorithm used for this research will remain constant.

(See <http://github.com/BenHocking/ShortCircuitGA> for the genetic algorithm source code.) The population size is 100, with 10 elites that are automatically propagated to the next generation without modification. Each genotype consists of 21 genes residing on the $[0, 1)$ interval, with each gene being mapped to the minimum and maximum values

allowed by the parameter it represents.

The mutation probability for each gene is 0.03, with a mutation being generated on the $[0, 1)$ value using a truncated normal distribution with a mean equal to the prior value of the gene, a standard deviation of 0.2, and bounds of $[0, 1)$. The approach used here is that when a gene is mutated to generate the new value from a normal distribution with a mean given by the unmutated value of the gene and a standard deviation of 0.2. All genes reside on the $[0, 1)$ number line and are mapped to a given minimum/maximum value using the formula $x(\text{max} - \text{min}) + \text{min}$ for continuous parameters and $\lfloor x(\text{max} - \text{min} + 1) \rfloor + \text{min}$ for discrete parameters.

Crossover is *uniform*, the probability that crossover will happen in a genome is 0.6, and the probability a particular gene will crossover (given that any crossovers happen) is 0.25. In the model used here, the location of a gene in the genotype bears no significance, so a uniform crossover scheme is employed. In this scheme (and with the given choices of selection and mating algorithm), an individual crossover probability of 0.25 is effectively the same as an individual crossover probability of 0.75.

The selection method used is *fitness proportionate selection* with an elitist group. The genes for the genotypes of the first population are all chosen using a truncated normal distribution with $\mu = 0.5$, $\sigma = 0.2$, and bounds of $[0, 1)$.

The genetic algorithm is run until a genotype in the population achieves target performance (see Table 5) or 100 generations are produced. A summary of the genetic algorithm is in Table 7.

Label	Name	Description
B_1	Target performance	$P(601,650) \geq \frac{1}{3} \frac{m_e}{a_d \cdot n_A }$
B_2	Poor performance	$\forall t \left[P(t, t + 49) < \frac{1}{6} \frac{m_e}{a_d \cdot n_A } \right]$
B_3	Good activity	$F_2 \geq \tau$ (See Table 8)

Table 5. Important behaviors in trace conditioning. Target performance is performance good enough to declare that the neural network has “acquired” trace conditioning. Poor performance describes neural networks that have not demonstrated movement towards acquiring trace conditioning, or possibly the network acquired trace conditioning previously and then lost it by trial 150. The term $\frac{m_e}{a_d \cdot |n_A|}$ is significant because it is approximately $E[P(651,750)]$ during training trials. B_1 and B_2 are calculated only on the last training trial, while B_3 is calculated on the first training trial and the first testing trial. B_2 trivially implies $\neg B_1$ (consider when $t = 601$). Determining good values for τ is a goal of this research. Cf. Table 6.

Desired activity rate (per ms): $a_d \in [0.0005, 0.0025]$ (or from 0.5 Hz to 2.5 Hz)
Percentage of neurons firing per second: $p_a = \frac{a_d}{\Delta t(\text{in seconds})}$ (50 to 250%)
Percentage of activity due to external activation: $p_e \in [0.2, 0.5]$
Number of externally activated neurons per time-step: $m_e = p_e n_A $
Size of the set of tone and puff neurons: $ n_t = 10m_e$ and $ n_p = 10m_e$
Number of neurons from set s firing between times t_1 and t_2 : $N(t_1, t_2, s) = \sum_{t=t_1}^{t_2} \sum_{i \in s} z_i(t)$
Fraction of neuronal activity due to puff neurons: $P(t_1, t_2) = \frac{N(t_1, t_2, n_p)}{N(t_1, t_2, n_A)}$
Average activity between times t_1 and t_2 : $A(t_1, t_2) = \frac{N(t_1, t_2, n_A)}{ n_A \cdot (t_2 - t_1 + 1)}$
Squared deviation from desired activity: $D(t_1, t_2) = (A(t_1, t_2) - a_d)^2$
Sample standard deviation of activity: $\sigma_s = \sqrt{\frac{1}{749} \sum_{t=1}^{750} (A(t, t) - A(1, 750))^2}$
Desired standard deviation of activity: $\sigma_d = \sqrt{0.015}$

Table 6. List of evaluating equations for a trace conditioning episode. n_t refers to the set of tone neurons, n_p refers to puff neurons, n_A refers to all neurons, m_e is the number of externally activated neurons in a time step, and a_d is the desired fractional activity. Cf. Tables 4, 5, and 8.

Genetic Algorithm Parameter	Value
Number of generations	Until target performance (see Table 5) or 50 generations
Population size	100
Number of elites	10
Gene type	Continuous on $[0, 1)$
Mutation probability	0.3
Mutation standard deviation	0.2
Total crossover probability	0.6
Crossover method	Uniform
Individual crossover probability	0.25

Table 7. A summary of parameters describing the genetic algorithm used in this research.

Fitness Function	Description
F_1 : Original	$k_{collapse}P(251,450) + k_{preblink}P(451,550) + k_{blink}P(551,650) + k_{bridge}P(651,750)$
F_2 : Proxy	$\frac{100}{\left(\sum_{c=0}^9 D(75c+1, 75(c+1))\right)} + \frac{a^2}{ \sigma_d^2 - \sigma_s^2 }$
F_3 : Short-circuit	$\begin{cases} F_2 & \text{if } F_2 < \tau_2 \\ \tau_2 + F_1 & \text{otherwise} \end{cases}$

Table 8. Fitness functions and their mathematical description. F_1 is the desired puff-neuron activity after 200 trials, while F_2 measures general network activity after the first trial. See Table 5 for additional explanation of performance and activity measurements. All fitness functions are non-negative over their domains. For F_1 , $k_{collapse} = 0.5$, $k_{preblink} = 0.8$, $k_{blink} = 1$, and $k_{bridge} = 0.5$.

Using this genetic algorithm configuration and the full fitness function F_1 (see Table 8), on the neural network described in the previous section, running to find a feasible parameter setting is prohibitively expensive. For example, a typical run required several days on a cluster of 30 computers. Although this is feasible if the sole purpose of the research is to find a single viable parameter setting, the time required for a single genetic algorithm to complete precludes *a thorough exploration of the parameter space and examination of the robustness of the solution*. Thus, in order to achieve those goals, the further research proposed here will focus on a more efficient fitness function.

One method for improving the efficiency of a genetic algorithm is through the use of a proxy [22]. One form of proxy used in genetic algorithms is an approximation, where

an expensive fitness function is replaced with a less expensive fitness function known to approximate the original [22, 23]. No known approximation exists for the problem being studied here, but there is a function that has an implicative relationship with the desired fitness function. Specifically, in the experiments I ran, I never saw good performance if activity was not good on the first training and testing trial. This can be restated as the implicative relationship that if there is not good activity on the first trials then there will be poor performance on the last trial (referred to as $\neg B_3 \Rightarrow B_2$ in Table 5).

Because determining whether good activity has occurred after one training trial only requires two trials to be run (one training and one testing), it is much less computationally expensive than determining whether good performance will occur, which requires 151 trials (150 training trials and a testing trial). *To take advantage of this significantly faster proxy function, I use a short-circuit fitness function (F_3) that combines a proxy function (F_2) with the full desired fitness function (F_1).* See Table 8 for a rigorous definition of these fitness functions.

Primary Fitness Function (F_1)

Any run of a genetic algorithm will involve a sampling of the parameter space. We call that sequence of samples the *genetic algorithm trajectory*. This experiment is intended to measure the evaluation efficiency along the genetic algorithm trajectory guided by the original fitness function (F_1). In this experiment the genetic algorithm is modified to calculate both F_1 and F_2 for all sampled parameter settings (*i.e.*, genotypes). For each run, $h^A(\tau, S_i)$ is computed for τ between 5×10^3 to 5×10^7 (as before), where S_i is the set of parameter settings for generation i . The genetic algorithm will be run at least seven times (with up to 4,510 fitness evaluations per run) to determine the confidence of this

measurement.

Short-Circuit Fitness Function (F_3)

When the genetic algorithm uses the short-circuit evaluation, the genetic algorithm trajectory will be different than when using the primary fitness function alone. This experiment is intended to measure the evaluation efficiency along the genetic algorithm trajectory guided by F_3 . Since F_3 depends on selected values of τ , we cannot use the full range of τ . As with the previous experiment, both F_1 and F_2 are calculated for all genotypes. The genetic algorithm will be run at least five times each for five different values of τ (based on interesting values of τ found in *IIA.2*), for a total of 25 different runs.

Determining the existence of parameter settings for acquiring trace conditioning

Determining whether a particular choice of parameter settings reliably acquires trace conditioning requires demonstrating several behaviors. First, the simulation must have activity of the puff neurons during the blink interval of at least one third that of the activity displayed during the puff interval of the prior training session. Second, the simulation must have less than half that activity during the collapse interval (the interval immediately after the tone interval). Third, the simulation must not show a collapse of activity during prior training intervals, as this would correspond to the individual experiencing brain death or a coma — something that does not happen in laboratory experiments that the simulations are intended to recreate. Finally, most simulations using the same settings but different random seeds should also satisfy the prior three conditions.

Proxy effectiveness

Calculating the fraction of episodes for which $\neg B_3 \Rightarrow B_2$ is true is intended to measure the proxy function's *effectiveness*. Currently, a value of 5×10^5 is being used for the threshold τ in evaluating B_3 . If we define $g^B(\tau, s)$ as 1 when $F_2(s) \geq \tau$ or B_2 (equivalent to $\neg B_3 \Rightarrow B_2$), and 0 otherwise, then *effectiveness* $h^B(\tau, S)$ is $\sum_{s \in S} g^B(\tau, s) / |S|$, where S is the set of parameter settings being evaluated.

The short-circuiting component of F_3 is most important for its “decision”: do you run the more computationally expensive objective function or not? As such, it suffers from the same problems as any inexact boolean test: false positives and false negatives. In this case, false positives result in us potentially missing useful parameter settings, and false negatives result in us wasting computational resources exploring useless parameter settings. Thus, in addition to demonstrating that the proxy has improved the efficiency of the genetic algorithm, the effectiveness of the proxy must be shown.

Measuring *proxy robustness* will help answer the question: can the genetic algorithm become even more *efficient* without becoming less *effective*? Examining ranges of τ from 5×10^3 to 5×10^7 , while measuring effectiveness will answer this question.

Evaluation efficiency

Evaluating the proxy fitness function requires only a single training trial instead of the 150 training trials required to evaluate the primary fitness function, so it will trivially require less computational time. Measuring this efficiency accurately requires calculating the computational time required to calculate the primary fitness function and the computational time required to calculate the proxy fitness function on the same computer.

Because most of the experiments are run on a grid computing system, this requires obtaining a representative sample and re-running them on a single computer.

Trajectory efficiency

To measure the *trajectory efficiency*, the ratio of the number of generations required to achieve B_1 along a genetic algorithm trajectory guided by F_3 to the number of generations required along a genetic algorithm trajectory guided by F_1 will be calculated. *Trajectory efficiency* is defined as 1 minus this ratio. Values less than zero demonstrate trajectory inefficiency. Initial experiments suggest that the proxy function provides a more productive *fitness landscape* in regions where B_3 is false resulting in fewer generations until target performance is achieved. One explanation for this improved fitness landscape is that for significantly poor activity control, the F_1 is zero (plus the addition of random noise), whereas F_2 is *not* constant, thus providing feedback for which parameter settings are better than others in episodes where the performance information is essentially useless.

Naïve exploration

Using a random sampling approach is intended to investigate evaluation efficiency over the parameter space without the biases introduced by the trajectory of a genetic algorithm. A straight-forward random sampling approach would be uniform sampling of each parameter in its $[0,1)$ mapped space. This has the disadvantage of disproportionately choosing samples far from the center of the domain space. Specifically, more than half of samples so chosen from a 21-dimensional space will be in either the $[0,0.02)$ or $[0.98,1)$ portion of the number-line for at least one parameter. Consider the unit square and unit hypercube. A square of one-half its area would have its sides equal to the square root of

one-half, or approximately 0.7071. Similarly, a 21-dimensional hypercube of one-half the hypervolume of a unit hypercube would have its edges equal to the 21st root of one half, or approximately 0.9675. Conversely, a square with side one-half is one-fourth the volume of the unit square, whereas a 21-dimensional hypercube of edge one-half is one-half to the 21st power, or approximately 4.768×10^{-7} the volume of the unit hypercube.

A way to address this is to instead represent points in the parameter space by a hyperball with random variables $(\rho, \varphi_1 \dots \varphi_{20})$ with ρ on the interval $[0,1)$, φ_1 on the interval $[0,2\pi)$, and $\varphi_2 \dots \varphi_{20}$ on the interval $[0,\pi)$; and then mapping the unit hyperball to the unit hypercube. The center of the hyperball will be mapped to an average over the location of target parameter settings (i.e., parameter settings meeting target performance) and the radius and directional vector of the point on the hyperball will map to the line segment connecting this point and where the directional vector would intersect with the boundary of the hypercube. One can imagine statistical biases introduced by this hyperball-to-hypercube mapping, so part of the research will involve either calculating the extent of these biases or finding yet a better sampling approach.

Each parameter setting sampling first shuffles the parameters so as to avoid bias among parameters. 15,000 parameter-setting samples will be generated by both approaches (uniform and hyperball), resulting in 3×10^4 random parameter setting samples.

Results

Improvement in biological plausibility

With the new extensions to the Levy model, we can directly map quantities in the model to neuron-level measurables. Specifically, for any neuron (or all neurons) we can determine what the membrane potential the model assigns to it at a millisecond resolution. Figure 5 gives an example of membrane potential versus time for a single neuron during the first training trial of trace conditioning. With McCulloch-Pitts neurons or even integrate-and-fire, there is no concrete neurophysiological meaning of the somato-dendritic excitation.

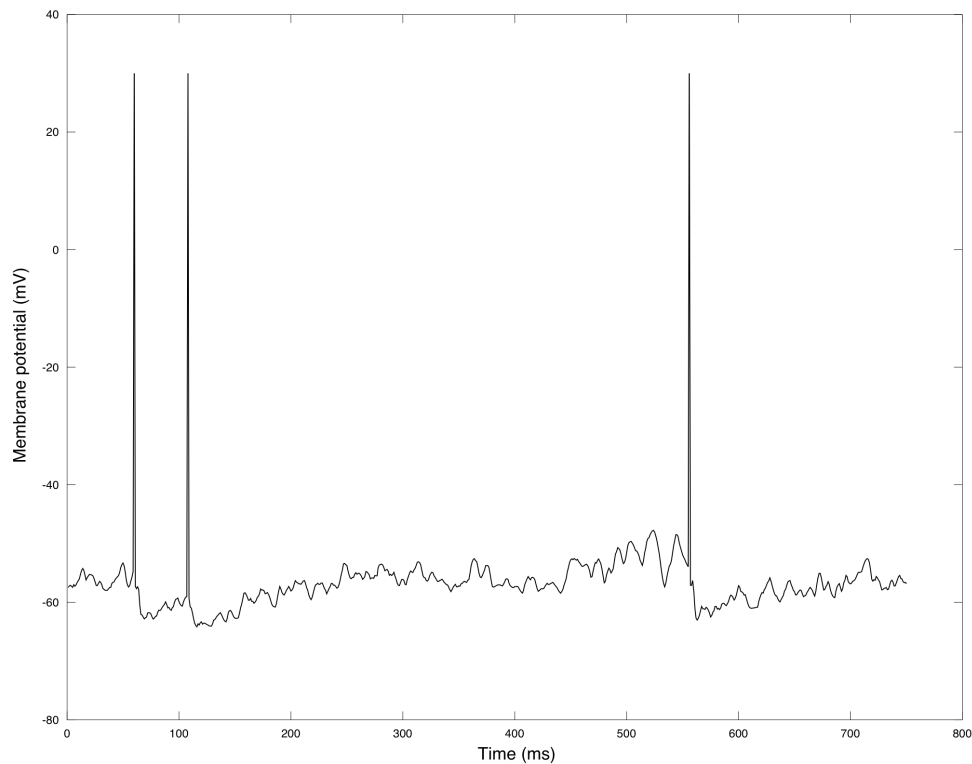


Figure 4. Membrane potential of a neuron during the first training trial of trace conditioning.

Gamma Oscillations

For the first gamma oscillation experiment, each pattern was presented to the network for 143 ms, so that one sequence of 32 patterns required a total simulated time of 4.576 seconds. After the network learns the sequence, synaptic modification is turned off, and the network is driven for another 500 trials, during which the activity of the network during each time-step (3.575 ms) is measured.

A portion of this activity is shown in Fig. 7A and B shows the resulting power spectrum. The lowest frequency spectral peak is 0.2 Hz. This maximum corresponds to the 4.576 s required to complete one iteration of the repeating sequence. Also visible are harmonics of this frequency, including the 32nd harmonic (7.0 Hz), which also corresponds to the pattern duration of 143 ms. Most importantly, there is a substantial increase in power distributed around 38.2 Hz, i.e. frequencies associated with gamma oscillations, which are present in the CA3–CA1 regions of the hippocampus (for example, [1]). Analogously, we will refer to power distributions near this frequency as gamma oscillations. These gamma oscillations differ from the other spectral peaks in the breadth of their distribution.

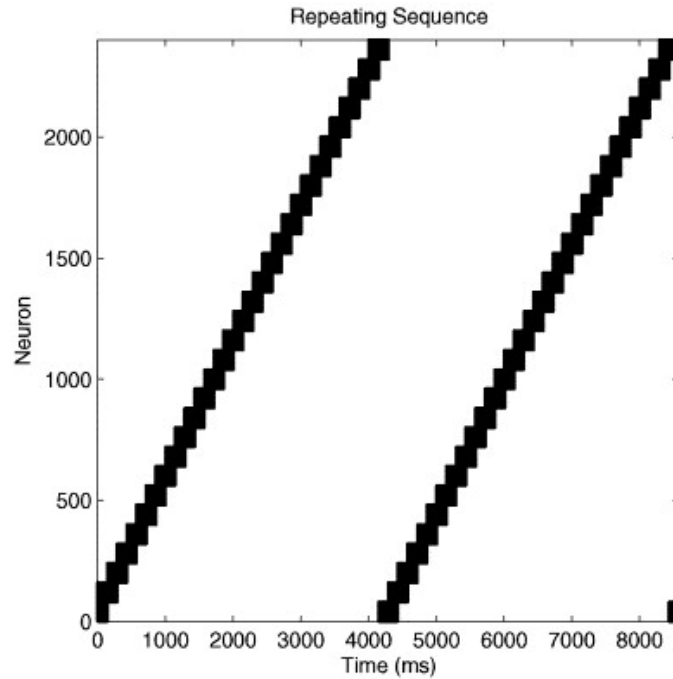


Figure 5. External input. A sequence of 32 patterns of externally activated neurons is presented repeatedly to a simulated neural network. Here two such presentations are shown. Each pattern consists of 160 externally activated neurons and has a duration of 143 ms. Successive patterns share 80 neurons. Only the first 2400 out of the entire 6000 neurons are part of the externally activated sequence.

When the repeating sequences of externally activated neurons are replaced with random input activation, the low frequency-oscillations associated with the length of the sequence disappear, but the gamma oscillations are still present with approximately the same center frequency (compare the top of Fig. 8 to the middle of Fig. 8). Finally, if all external input is removed and inhibition is adjusted to maintain the same 2 Hz average firing rate, the same results are produced—the center frequency of the gamma oscillations remains near 40 Hz (Fig. 8, bottom).

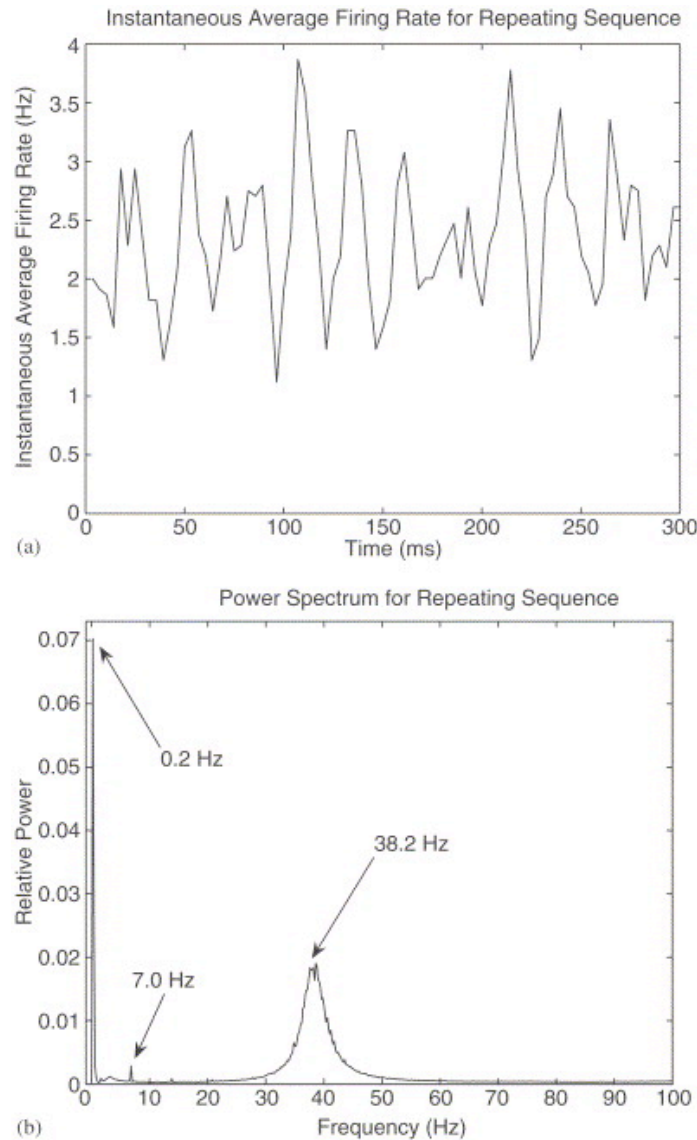


Figure 6. A gamma oscillation appears in the macroscopic activity of the network. The instantaneous firing rate shown here is calculated by dividing the number of neurons that fire in each time-step by the size of the time-step (3.575 ms). The 12 major cycles during this 300 ms window leads to a rough estimate of 25 ms per oscillation, or a frequency of approximately 40 Hz. (B) Power spectrum of instantaneous firing rates (see A) after training, averaged over 500 trials, with each trial lasting 4.576 simulated seconds. The network is trained on a repeating sequence until the sequence is learned. Then, the sequence is driven with this same sequence for 500 trials with no synaptic modification, and the power spectrum of the total network activity for each time-step of the simulation is obtained. The data set used to generate the power spectrum consists of 640,000 values.

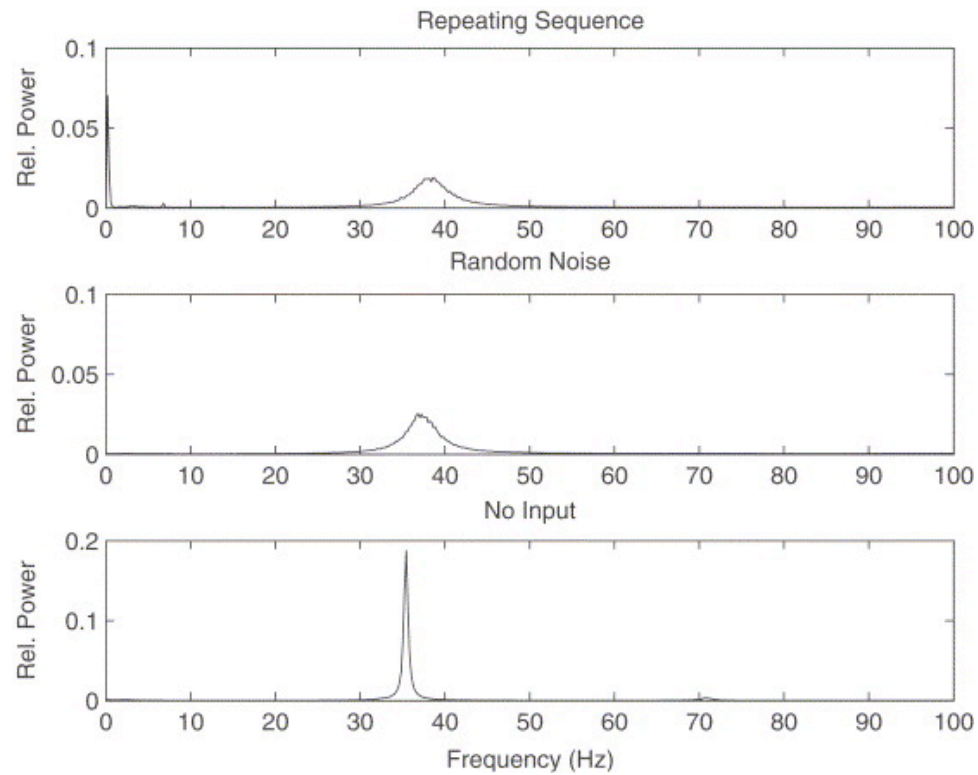


Figure 7. Power spectrum for simulations with an input of a repeating sequence (top), with random input (middle), and with no external input (bottom). Note that the vertical scale on the bottom figure is twice that of the other two figures.

Figure 9 shows the results of changing average firing rates of neurons across a range, from 2 up to 8 Hz. As can be seen from Figure 9, the network experiences a continuous-phase transition near 4 Hz as the gamma oscillations are replaced with harmonics of the average firing rate. For example, for simulations with an average firing rate of 8 Hz, the lowest peak in the power spectrum is 8 Hz, and the remaining peaks are harmonics of 8 Hz.

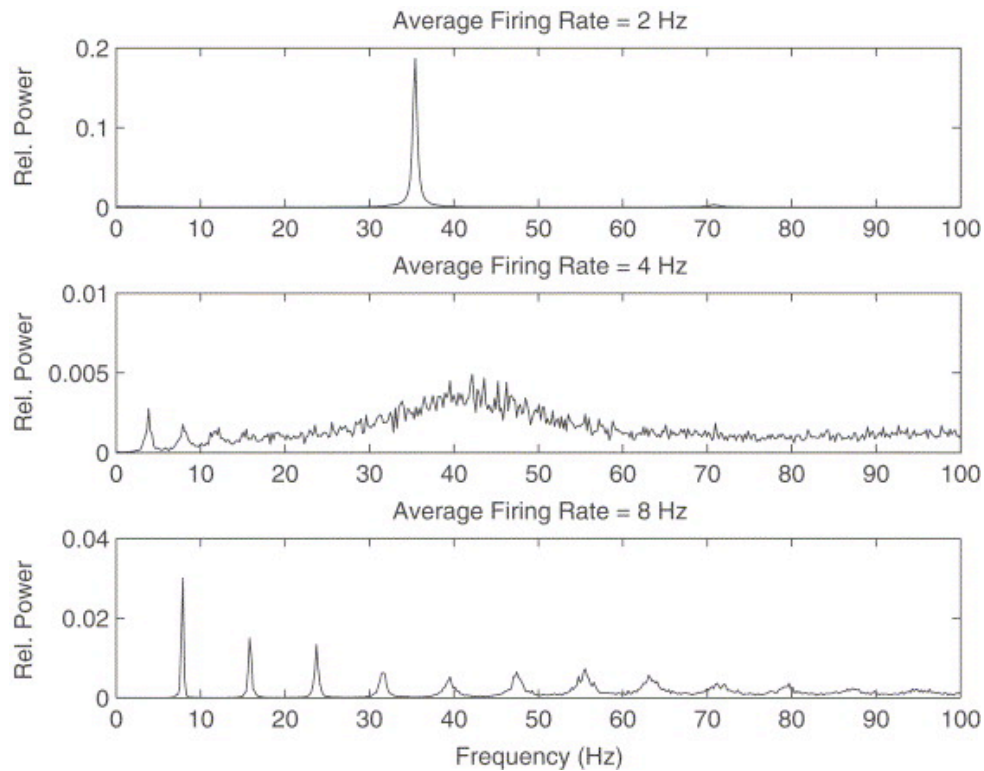


Figure 8. Power spectrum for simulations with an average firing rate of 2 Hz (top), 4 Hz (middle), and 8 Hz (bottom). At 2 Hz, the gamma oscillations are prominent, but at 4 Hz, their power is much weaker. At 8 Hz, the gamma oscillations are lost, replaced with a spectral peak at 8 Hz and its harmonics. Interestingly, the harmonics are not monotonically decreasing in peak power; note the dip in the harmonics near 40 Hz.

Fig. 10A and B show reordered firing diagrams for the networks corresponding to the top and bottom graphs in Fig. 4, respectively. When a simulation runs at 2 Hz, the average network activity (for an untrained network) shows little evidence of synchronized firing. This lack of synchrony is one feature that sets it apart from models such as the one used in Kopell et al. [3]—none of the neurons in the network are firing at the gamma frequency, the gamma oscillations only exist in the network as an ensemble. However, for simulations run at 8 Hz, the networks do show significant, albeit approximate, synchrony. In Fig. 10, the gamma frequency is visible in the reordered firing diagram. This change in the tendency to synchronize firing demonstrates behavior similar to a

continuous phase transition and this occurs at around an average firing rate of 4–5 Hz in these simulations.

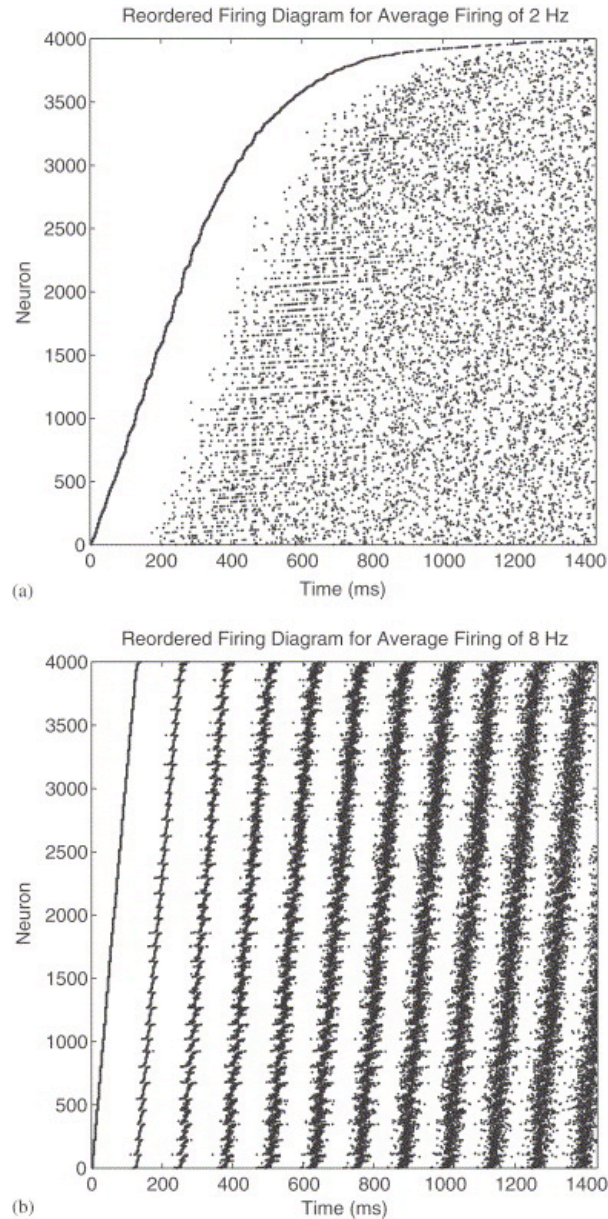


Figure 9. (A) A reordered firing diagram for a simulation run at 2 Hz. The neurons are reordered based on when they first fire. At 2 Hz there is not a significant amount of synchronous behavior present, although macroscopic gamma oscillations exist. (B) A reordered firing diagram for a simulation run at 8 Hz. This firing diagram is reordered in the same manner as in Fig. 5A. In this diagram, neurons demonstrate synchronous behavior. The apparent shift across time from very ordered firing to less ordered firing is due to the method used for reordering.

Theta Modulated Gamma Oscillations

In the absence of any theta-modulation, gamma oscillations were quite strong and centered around 50 Hz. Fig. 12a shows the FFT of the net activity, and Fig. 12b shows the log of the FFT converted to dB, i.e., multiplied by 10.

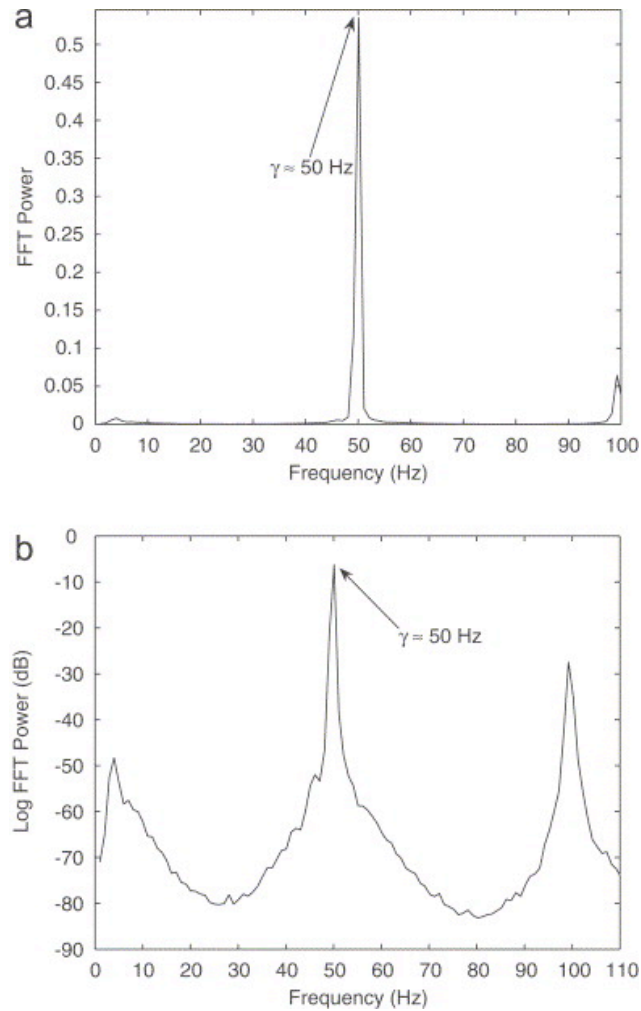


Figure 10. Power spectrum of a simulation in the absence of any external input demonstrating emergent gamma oscillations. When the network is allowed to run without any input, a very sharp gamma oscillation is present, as is its harmonic. A low-powered oscillation is also visible at approximately 3–4 Hz, which is more noticeable in Fig. B. In both figures, the absolute power in the FFT is normalized to 1.

When this same network is trained on theta-modulated external input and then allowed to run in the absence of any input, a very low-power oscillation in the theta band is now visible (Fig. 13). Additionally, the central frequency of the gamma oscillation has shifted from 50 Hz to approximately 64 Hz, and the gamma frequency oscillation is more spread.

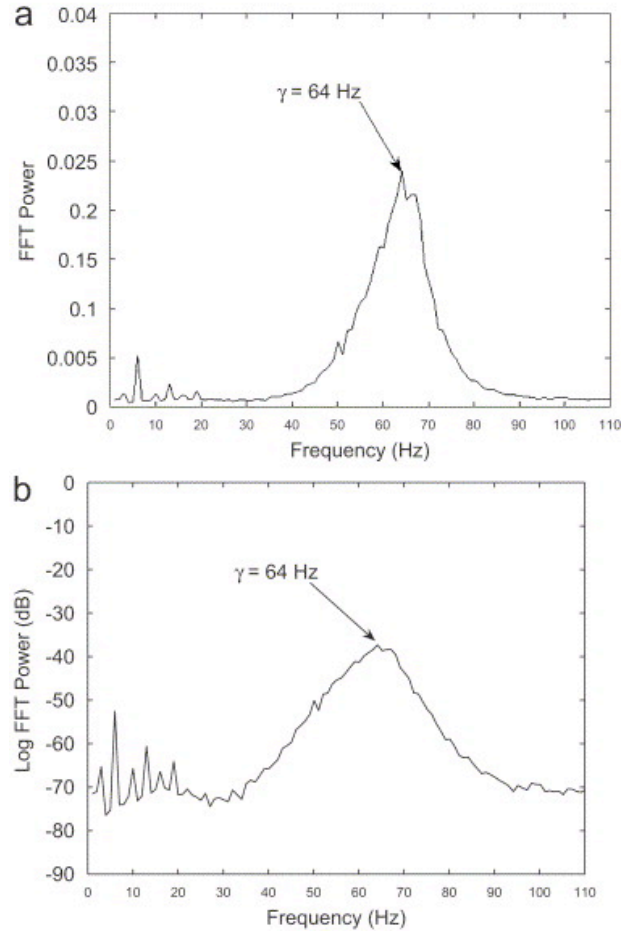


Figure 11. Power spectrum in a simulation without external input after training on theta-modulated input. The FFT shown here is for the same network as in Fig. 12, but after training the network using theta-modulated input. In both figures, the absolute power in the FFT is normalized to 1.

If instead, the network is driven at theta-modulated input, the gamma oscillations become much less powerful. Although not visible on a linear scale, Fig. 14a, the log of

the FFT reveals that the gamma oscillations are still present, but are several orders of magnitude weaker than in the absence of theta-modulated input (Fig. 14b).

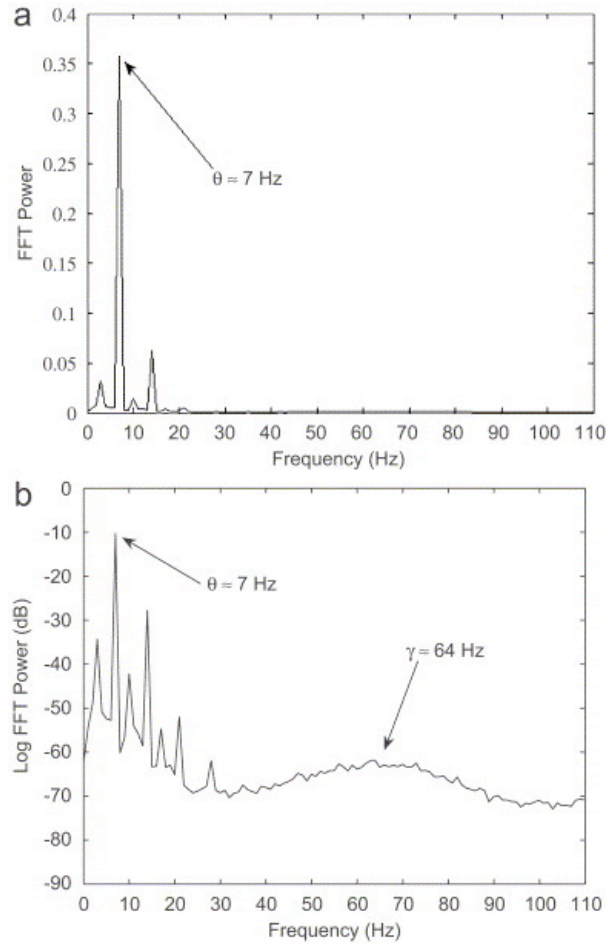


Figure 12. Power spectrum for simulations in the presence of a theta-modulated input. The log of the FFT reveals that gamma still exists but is more than 100 times weaker than in the absence of theta-modulated input. In both figures, the absolute power in the FFT is normalized to 1.

Existence of Settings for Trace Conditioning Acquisition

Multiple settings were found that allowed simulations to acquire trace conditioning without activity collapse for any training trials and for multiple random seeds. An example of trace conditioning acquisition can be seen in Figure 15. Activity of the neurons encoding the puff/blink response goes up prior to the interval when the puff had been delivered during training, giving the individual sufficient time to blink.

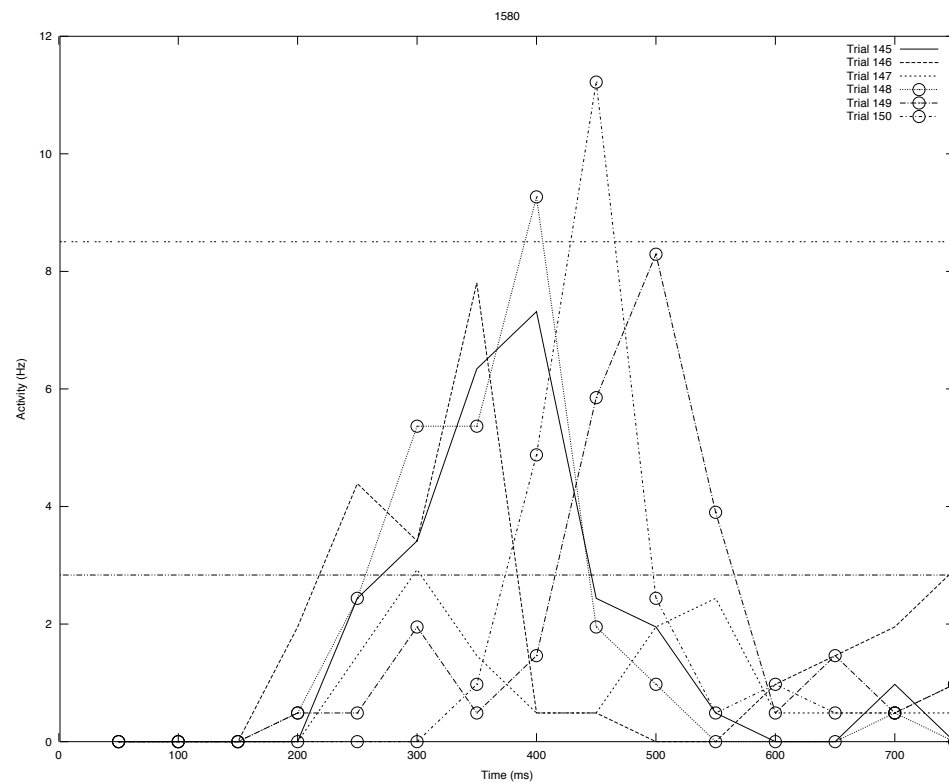


Figure 13. Trace conditioning acquisition.

If we examine across trial activity over the entire range of training (Fig. 16), then we see that although activity decreases, it never collapses. Similarly, examining within trial activity for individual trials shows an activity decrease during the trace interval, but no collapse.

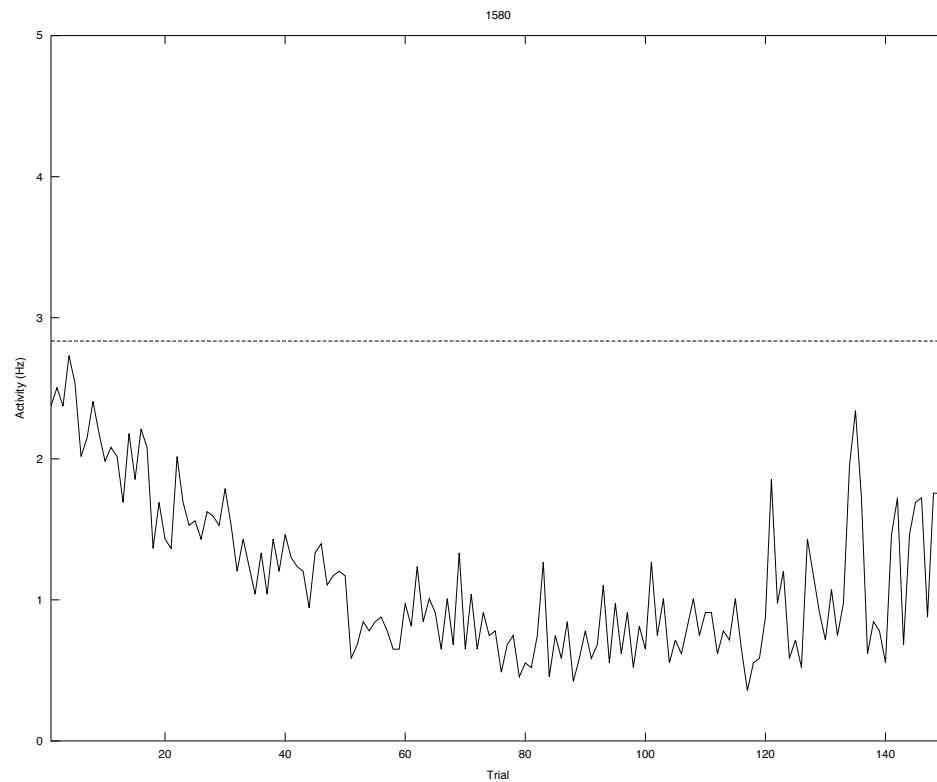


Figure 14. Across trial activity for an individual simulation that demonstrated trace conditioning acquisition.

Finally, we examine the same settings but for multiple random seeds. Figure 17 shows trace conditioning acquisition when the random seed is set to 1, and other random seed settings can be found in the appendix.

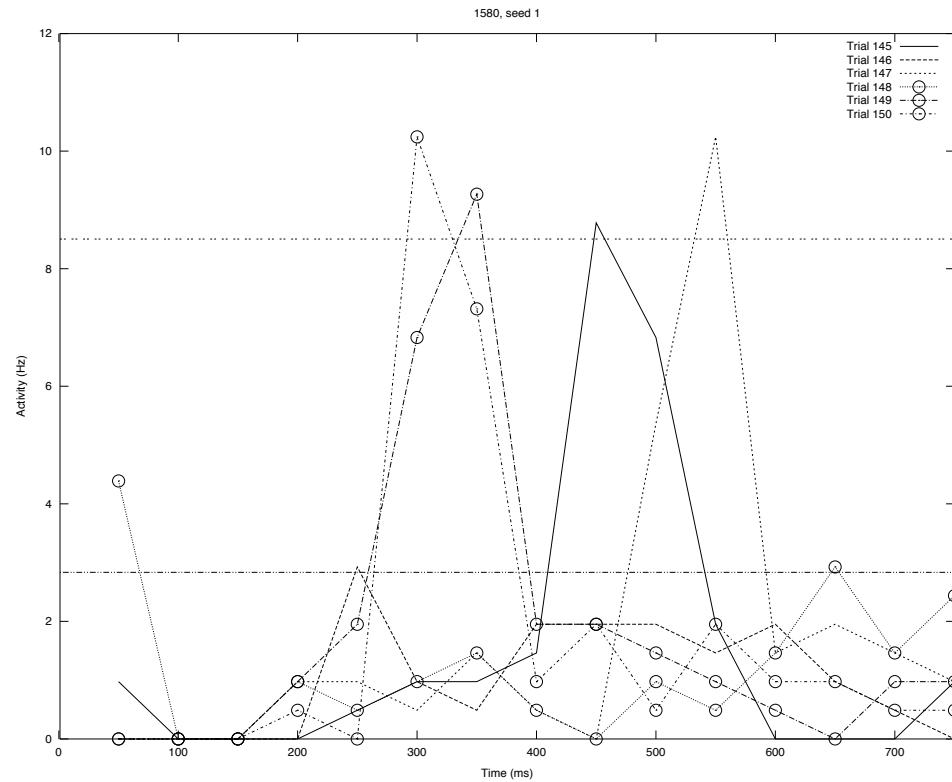


Figure 15. Trace conditioning acquisition for an individual using parameter settings discovered during a genetic algorithm experiment, but with a different random seed than was used during that experiment.

Proxy effectiveness

One measure of proxy effectiveness is shown in Figure 18 as a plot of proxy threshold (the value used to compare against the proxy fitness function to determine whether evaluation of the primary fitness function is required) versus false negatives to determine how reliable is the proxy threshold at predicting an implicative relationship between the proxy fitness function and the primary fitness function.

A secondary measure of proxy effectiveness is shown in Figure 19 (see also Fig. 20) as a plot of proxy threshold versus false positives to determine how valuable the

proxy threshold is at eliminating unnecessary evaluations of the primary fitness function. As with all false positive and false negative measures, these two measures trend in opposite directions. The false negative measure is strictly non-decreasing whereas the false positive measure is strictly non-increasing.

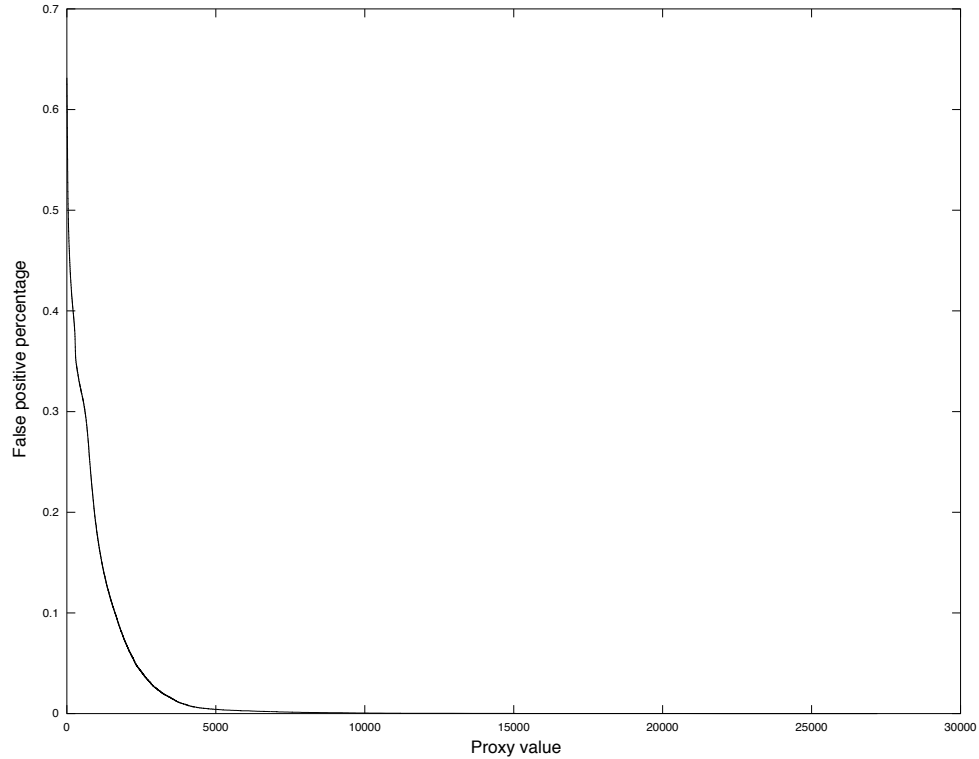


Figure 16. In a genetic algorithm using the short circuit fitness function, false positives as a function of the proxy threshold value, τ . As τ increases, more simulations fail to reach the threshold so that fewer evaluations of the primary fitness function are required. Since most of the simulations do not acquire trace conditioning, increasing τ usually decreases the false positives.

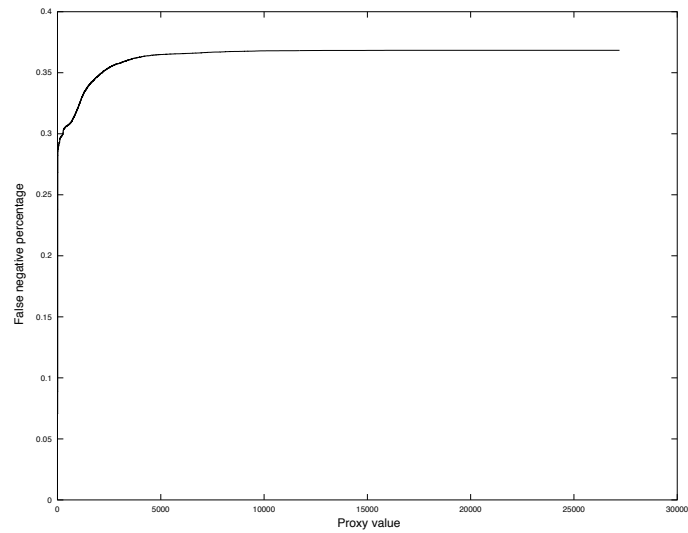


Figure 17. In a genetic algorithm using the short circuit fitness function, false negatives as a function of the proxy threshold value, τ . As τ increases, more simulations fail to reach the threshold so that fewer evaluations of the primary fitness function are required. Eventually, individuals that would have acquired trace conditioning are not considered, increasing the false negative rate.

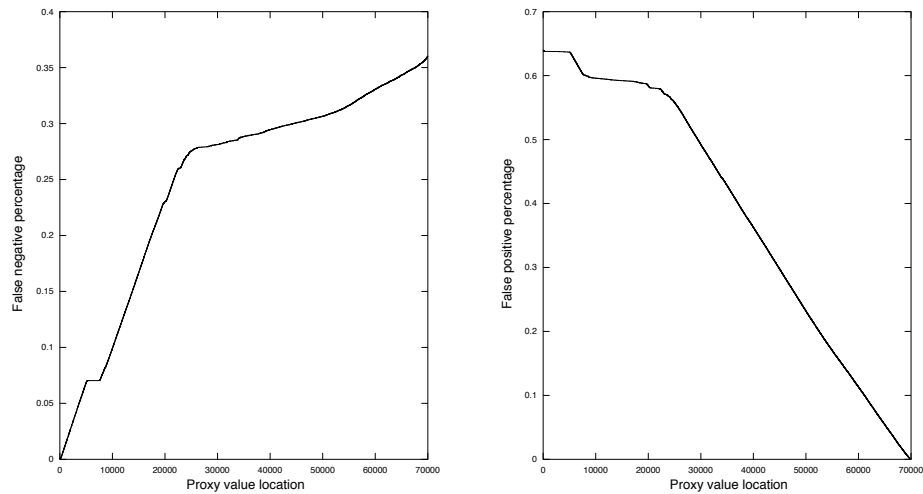


Figure 18. In a genetic algorithm using the short circuit fitness function, false negatives (left) and false positives (right) as a function of the proxy value's relative position.

Because there are far fewer simulations that demonstrate target behavior (i.e., trace acquisition) than merely non-poor behavior (i.e., learning to activate the puff neurons at some point after the tone neurons are activated), I also analyze how well the proxy fitness function predicts non-poor behavior. Figures 20 and 21 show these results.

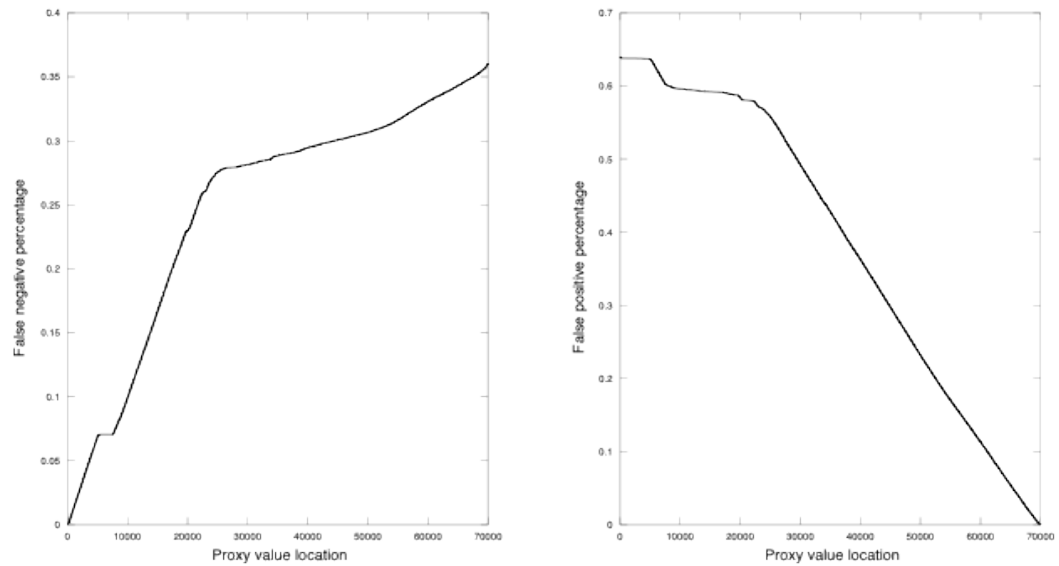


Figure 19. In a genetic algorithm using only the primary fitness function, false positives as a function of the proxy threshold value, τ . As τ increases, more simulations fail to reach the threshold so that fewer evaluations of the primary fitness function are required. Since most of the simulations do not acquire trace conditioning, increasing τ usually decreases the false positives.

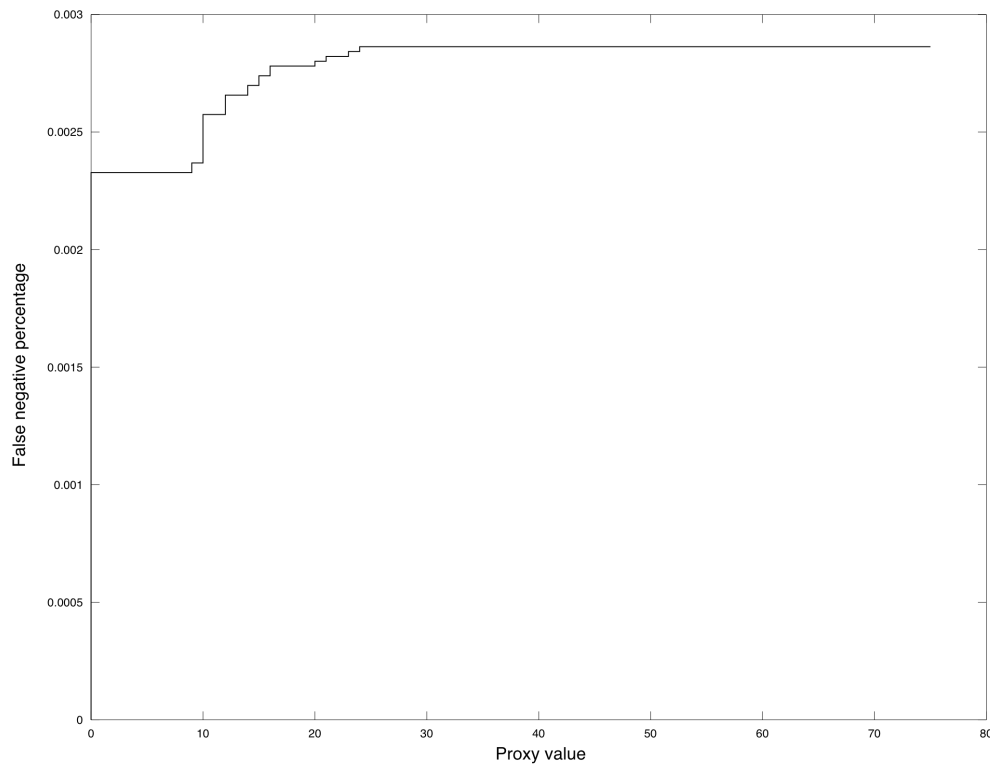


Figure 20. In a genetic algorithm using only the primary fitness function, false negatives as a function of the proxy threshold value, τ . As τ increases, more simulations fail to reach the threshold so that fewer evaluations of the primary fitness function are required. Eventually, individuals that would have acquired trace conditioning are not considered, increasing the false negative rate.

Evaluation efficiency

Evaluation efficiency of the short circuit fitness function is a combination of two factors: what fraction of the time does the proxy fitness function take to compute relative to the primary fitness function, and what fraction of the evaluations does the proxy fitness function identify as not requiring the primary fitness function to be calculated.

The first factor can be expected to be fairly constant, as regardless of the proxy threshold the number of trials required to calculate the proxy fitness function is always

one, and the number of trials required to calculate the primary fitness function is 150. However, there is a non-trivial set-up cost dominated by the amount of time required to construct the random topology of the simulation that is incurred regardless of the number of training and testing trials that will be run in the simulation. This set-up computational cost is independent of neural activity (i.e., how many neurons fire per time step), but the computational cost of each trial is not. When activity control is good (which is what the proxy fitness function is a measure of), one can expect a relatively constant speedup of the proxy fitness evaluation over the primary fitness function. However, when activity is much lower than desired, the speedup of the proxy fitness evaluation will be markedly less because each trial will require less time while the setup cost will remain approximately constant, and conversely when activity is much higher than desired, the speedup of the proxy fitness evaluation will be markedly higher because each trial will require more computational time. Figure 22 shows that during the first generation of the genetic algorithm using the short-circuit evaluation the speedup is much greater than during subsequent generations, when activity control is better. Thus, the higher activity simulations have more of a speedup increase than the lower activity simulations have of a speedup decrease (relative to the steady-state speedup). This is not surprising because the desired activity is to have only 0.2% of the neurons firing on every time step—there is much more opportunity for neurons to fire more frequently than this target than there is for neurons to fire less frequently. Figure 23 shows the same data for genetic algorithm runs using only the primary fitness function. Activity control is not part of the primary fitness function, and this is reflected in the theoretical speedup being much higher for many generations. In this scenario, a higher speedup reflects a negative situation—the

primary fitness function takes much longer to calculate because of neural activity that is much higher than desired.

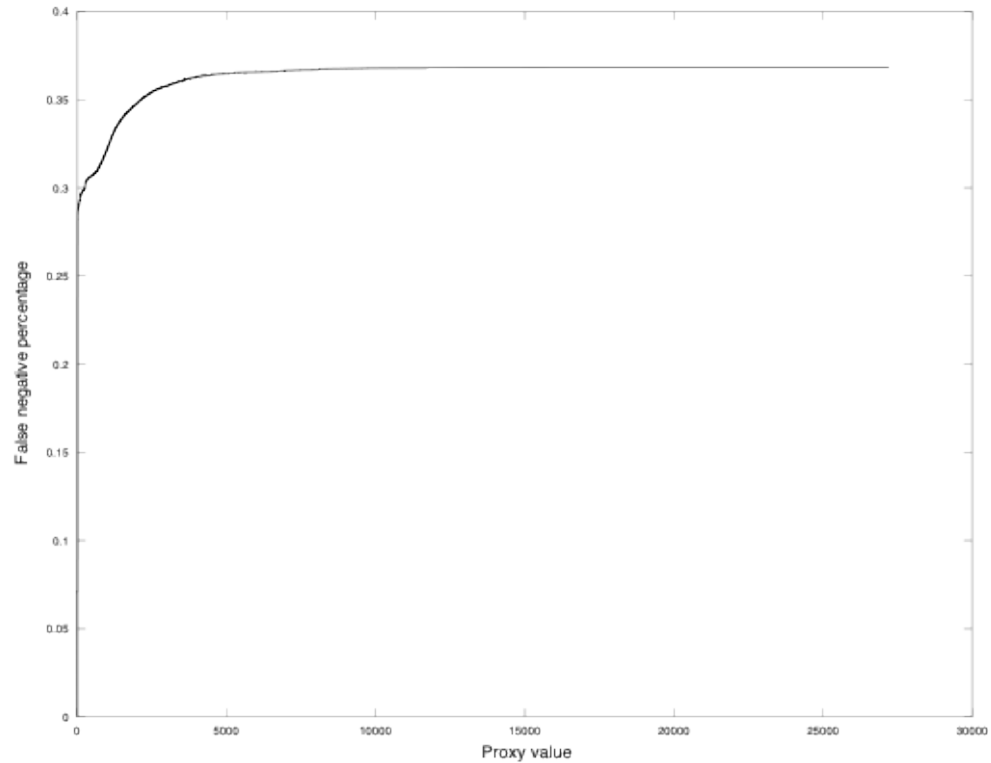


Figure 21. Speedup of simulation calculating only proxy fitness function over simulation calculating primary fitness function for individuals in generation for genetic algorithms using the short circuit fitness function. See text for discussion of why the speedup is higher in the first generation than in subsequent generations. Only the first 15 generations are shown. The full 100 generations can be seen in the appendix.

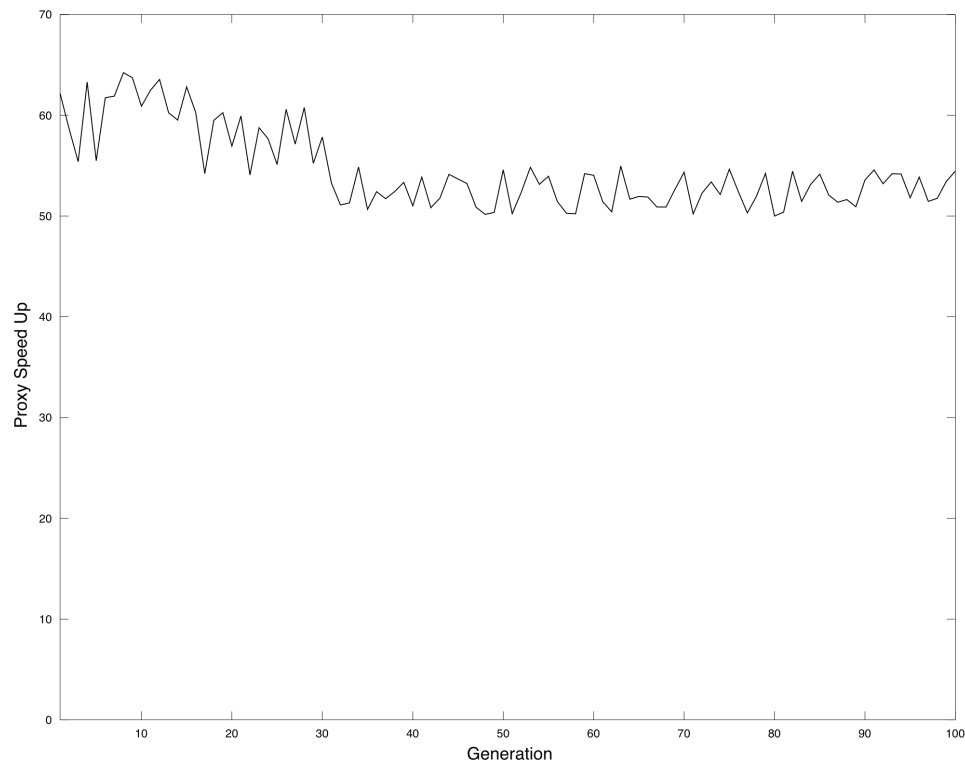


Figure 22. Speedup of simulation calculating only proxy fitness function over simulation calculating primary fitness function for individuals in generation for genetic algorithms using only the primary fitness function.

The second factor is shown in Figure 24 where the fraction of simulations that did not require computation of the primary fitness function is plotted per generation, averaged over all genetic algorithms using the short-circuit evaluation approach. Figure 25 evaluates the fraction of simulations that theoretically would not have required computation of the primary fitness function for genetic algorithms that used only the primate fitness function. However, as Figure 24 demonstrates, using a fitness function that rewards having to calculate the primary fitness function predictably leads to fewer simulations that do not need to calculate the primary fitness function. Thus, after multiple

generations, the evaluation efficiency of the short-circuit genetic algorithm is significantly reduced.

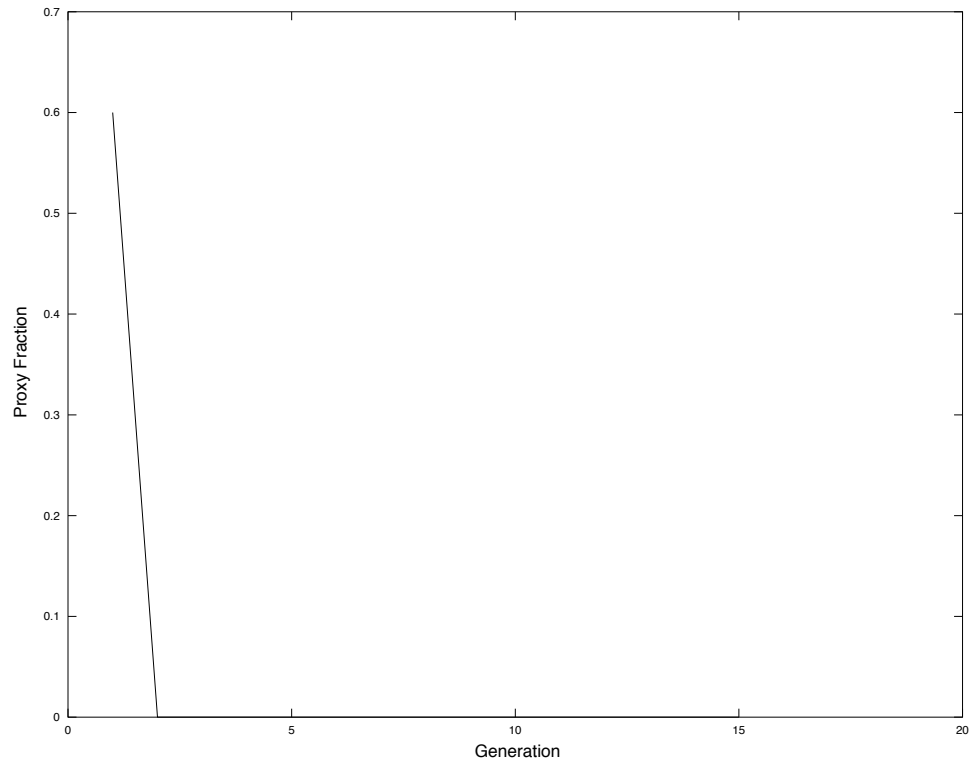


Figure 23. In a genetic algorithm using the short circuit fitness function, the fraction of simulations that did not require evaluating the primary fitness function.

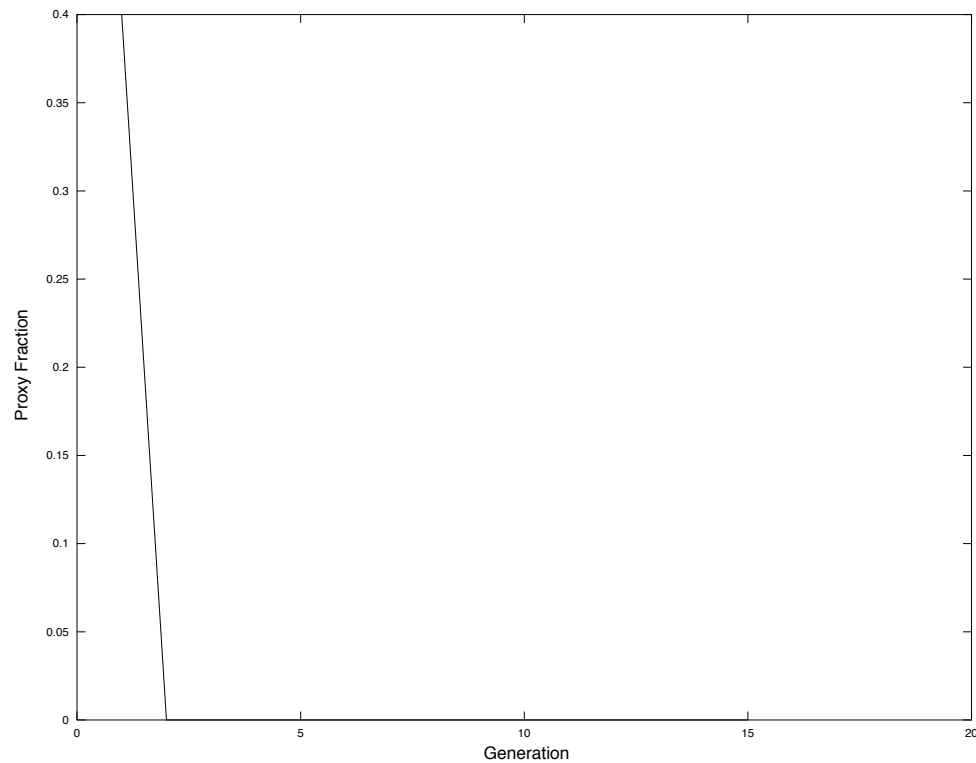


Figure 24. In a genetic algorithm using only the primary fitness function, the fraction of simulations that did not require evaluating the primary fitness function.

Trajectory efficiency

Genetic algorithms using the short-circuit fitness function discovered simulations acquiring trace conditioning behavior in far fewer generations than genetic algorithms using only the primary fitness function. Three of the seven genetic algorithms that were run using only the primary fitness function never found simulations that acquired trace conditioning, and the other four took an average of 67.5 generations before a simulation was discovered meeting the necessary criteria. By comparison, genetic algorithms using the short-circuit genetic algorithm found simulations acquiring trace conditioning within 25.3 generations on average, with one genetic algorithm run requiring only 9. If we

eliminate the one outlier requiring only 9 generations, the remaining 6 genetic algorithm runs required an average of 28 generations. Even discarding the three genetic algorithms running only the primary fitness function that never found a solution, the short circuit genetic algorithms demonstrate a trajectory efficiency of 58.5%. If we assume that the three discarded genetic algorithm runs would have found a solution on the 101st generation (increasing the 67.5 average up to 81.9), and do not eliminate the short circuit genetic algorithm outlier that found a solution in only 9 generations, then the trajectory efficiency is 69.1%.

Conclusions

Contributions

Extension to Levy Model and Existence Proof of Viable Parameters

The potential benefits of the Levy model are that it can aide both in understanding the mammalian hippocampus as well as in providing inspiration for new approaches in artificial intelligence. This model will be able to improve our understanding of the mammalian hippocampus by forming a bridge across temporal timescales between the research already done with the Levy model for multiple behavioral and training tasks[7] and research done with more realistic models of pyramidal neurons. As a proof of concept, the extended model has been demonstrated to work on a simple network at the nanosecond timescale (i.e., at timescales shorter than previous research by a factor of over 10 million). This was a toy problem and was not intended to demonstrate the feasibility of running large networks at nanosecond timescales, but rather to demonstrate

that the simulations were robust with respect to changing the size of the time step. This was not true of previous versions of the Levy model. Some of the specific neurological benefits that this model could contribute are the screening of existing neurotropic drugs, the design of new neurotropic drugs, and possibly new non-drug related treatments of brain disorders involving the hippocampus, such as post-traumatic stress disorder[LDA03]. A model of the hippocampus that can accurately model the hippocampus across multiple timescales and with multiple levels of detail could be a starting point from which to alter parameters according to how an existing neurotropic drug is thought to act, either through the use of a domain expert, or through the use of further metaheuristics. Alternatively, if a particular system-level change in the hippocampus is desired, the model might be able to provide insight into what lower-level changes are required, as well as what possible side-effects might accompany that change. Such knowledge would be an asset in designing new drugs.

In addition to the benefits of understanding the mammalian brain, discoveries made using the Levy model can provide inspiration for new approaches in artificial intelligence. For example, simulations of the Levy model incorporating synaptic failures demonstrated that these failures improved simulations' performance on the transverse patterning task, and that the optimal amount of failure increased with larger neural networks. [8]

Effective and Efficient Technique for Discovery

Although using a proxy to increase the efficiency of genetic algorithms is not novel [22, 23], this research shows the utility of using implicative relationships to craft proxy functions in the absence of an approximating fitness function. The proxy technique

here differs from other approaches in that the proxy function is not intended to approximate the value of the expensive fitness function it is a proxy for, but rather is intended to predict whether that expensive fitness function will exceed a certain minimum threshold. The differences between using non-approximating proxy functions and more traditional proxy functions provide both advantages and disadvantages, but the similarities are arguably more significant than the differences.

The primary difference between non-approximating proxy fitness functions and approximation fitness functions is that non-approximating fitness functions are not required to provide a reasonable approximation that would be returned by the fitness function they are a proxy for. This has the disadvantage of limiting the proxy use of the non-approximating function to only a portion of the domain space spanned by the function being optimized, but it has the advantage of allowing the use of a proxy function in cases where there is no known valid approximation.

The primary similarity between non-approximating and approximation fitness functions is that they both act as a proxy, thus allowing a less expensive calculation to stand in for its more expensive counterpart. The advantage of such proxies is clear: given a fixed set of resources, the problem domain can be explored more quickly and/or more thoroughly. A secondary similarity between non-approximating proxy fitness functions and approximation fitness functions is that a domain expert might sometimes be required to help identify where, how, and which such proxies can be used. Alternatively, techniques used to automatically find approximation fitness functions [22] could be adapted to find non-approximating proxy fitness functions.

Future Work

Several interesting questions arise as part of this work:

1. What changes can be made to the hippocampal model to make it even more neurophysiological without adding too much complexity or computational cost?
2. Can the same approach be used to find parameter settings that reproduce configural learning?
3. Can the genetic algorithm be improved to counteract premature convergence?
4. Are there ways that the genetic algorithm analysis can be applied to other non-related research topics?

In looking to make the model even more neurophysiological, there are simple changes that add significant computational cost, and computationally inexpensive changes that might add significant complexity. In the former category, a simple change that might be desired is the use of more realistic theta oscillations as described in the section on theta-modulated gamma oscillations. The reason these are currently computationally expensive is that the theta modulation is done through the NeuroJet scripting language (see the appendix for a sample script). If the NeuroJet executable were enhanced to allow for the script to select the more realistic theta modulation, then much of the computational overhead would be avoided. Another simple change that is computationally expensive is to pre-train the neural networks on other non-related stimuli prior to training on trace conditioning. In lab experiments that are being modeled, the rabbits have had many prior

experiences before being trained on trace conditioning. An example of a complex change that does not require noticeable increases in computational time is the use of variations on the dendritic filter function that modulates the input from the dendrite to the soma. It is currently implemented as a look-up table such that any arbitrary function of time can be specified (with one entry per time interval), so it would require no additional computational time for other dendritic filter functions to be used. Determining which dendritic filter function to use, however, can be a complex proposition, whether we allow the genetic algorithm to select from a range of options or we use the medical literature.

There should be no reason why the same approach cannot be used on configural learning problems such as transverse patterning or transitive inference. Ideally, we would like to find a single set of neural network parameters that allow a simulation to learn trace conditioning, transverse patterning, and transitive inference. This could be tried using either the extended model discussed here or with the simpler model used elsewhere.

In the genetic algorithms using only the primary fitness function, 3 of the 7 genetic algorithm runs never found a solution that acquired trace conditioning. As we know from other runs there is a solution, these runs most likely converged prematurely. Approaches that might be used to prevent premature convergence include using multiple sub-populations and increasing the mutation rate.

Perhaps the most important feature of the short circuit genetic algorithm was its improvement of trajectory efficiency. We might expect to find similar improvements in other domains where the solution space has properties similar to

the properties of the solution space here. Key features include parameters that are strongly connected and an objective function that is relatively flat when the parameters being explored are not near a good solution. In this case, we have many parameters that affect network activity, network activity is poor over much of the domain, and when network activity is poor we gain little information from the primary fitness function.

References

- [1] Levy, William B (1996). *A Sequence Predicting CA3 is a Flexible Associator That Learns and Uses Context to Solve Hippocampal-like Tasks*. Hippocampus, 6, 579-590. ([http://faculty.virginia.edu/levylab/Publications/script/Hippocampal/HippoSpatial/Levy 1996 Hippocampus.pdf](http://faculty.virginia.edu/levylab/Publications/script/Hippocampal/HippoSpatial/Levy%201996%20Hippocampus.pdf))
- [2] Hodgkin, A., and Huxley, A. (1952). *A quantitative description of membrane current and its application to conduction and excitation in nerve*. Journal of Physiology, 117(4), 500–544. (<http://www.ncbi.nlm.nih.gov/pmc/articles/PMC1392413>)
- [3] FitzHugh, R. (1961). *Impulses and physiological states in theoretical models of nerve membrane*. Biophysical Journal, 1(6), 445-466. ([http://dx.doi.org/10.1016/S0006-3495\(61\)86902-6](http://dx.doi.org/10.1016/S0006-3495(61)86902-6))
- [4] Nagumo, J.; Arimoto, S.; Yoshizawa, S. (1962). *An active pulse transmission line simulating nerve axon*. Proceedings of the Institute of Radio Engineers. 50(10), 2061–2070. (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4066548)
- [5] Izhikevich, E.M. (2003). *Simple model of spiking neurons*. IEEE Transactions on Neural Networks, 14, 1569-1572. (<http://www.nsi.edu/users/izhikevich/publications/spikes.pdf>)
- [6] Izhikevich, E.M. (2004). *Which model to use for cortical spiking neurons?* IEEE Transactions on Neural Networks, 15, 1063-1070. (<http://www.nsi.edu/users/izhikevich/publications/whichmod.pdf>)
- [7] Levy, William B; Hocking, Ashlie B.; & Wu, Xiangbao. (2005). Interpreting hippocampal function as recoding and forecasting. Neural Networks, 18(9),1242-1264. (<http://www.sciencedirect.com/science/article/pii/S0893608005001917>)

- [8] Sullivan, David W. and Levy, William B (2004). *Quantal synaptic failures enhance performance in a minimal hippocampal model*. Network: Computation in Neural Systems, 15(1), 45-67.
(<http://taylorandfrancis.metapress.com/link.asp?id=mm00842552040623>)
- [9] Levy, William B and Wu, Xiangbao (2000). *Some Randomness Benefits a Model of Hippocampal Function*. In H. Liljenstrom, P. Arhem, & C. Blomberg (Eds.), *Disorder Versus Order in Brain Function*, 221-237. Singapore: World Scientific Publishing. (<http://books.google.com/books?id=BX5i6B5P3QEC&pg=PA221>)
- [10] Levy, William B; Colbert, C. M.; Desmond, Nancy L (1990). *Elemental adaptive processes of neurons and synapses a statistical/computational perspective*. M. Gluck, D. Rumelhart (Eds.), *Neuroscience and Connectionist Theory*, Lawrence Erlbaum Associates, Hillsdale, New Jersey, 187–235 (Chapter 5).
- [11] Hughes, J. R. (2008). *Gamma, fast, and ultrafast waves of the brain: their relationships with epilepsy and behavior*. Epilepsy Behavior 13(1), 25–31.
- [12] Vanderwolf, C.H. (2000). *Are neocortical gamma waves related to consciousness?* Brain Research 855(2), 217–224.
- [13] Gold, Ian (1999). *Does 40-Hz oscillation play a role in visual consciousness?* Consciousness and Cognition, 8(2), 186–195.
- [14] Bland, B. H.; Oddie, S. D. (2001). *Theta band oscillation and synchrony in the hippocampal formation and associated structures: the case for its role in sensorimotor integration*. Behavioral Brain Research, 127(1–2), 119–136.
- [15] Whishaw, I.Q.; Vanderwolf, C. H. (1973). *Hippocampal EEG and behavior: changes in amplitude and frequency of RSA (theta rhythm) associated with*

- spontaneous and learned movement patterns in rats and cats*. Behavioral Biology, 8(4), 461–484.
- [16] Cantero, J.L.; Atienza, M.; Stickgold, R.; Kahana, M. J.; Madsen, J. R.; Kocsis, B. (2003). *Sleep-dependent theta oscillations in the human hippocampus and neocortex*. Journal of Neuroscience, 23(34), 10897–10903.
- [17] Fabricius, Katrine; Wörtwein, Gitta; Pakkenberg, Bente (2008). *The Impact of Maternal Separation on Adult Mouse Behavior and on the Total Neuron Number in the Mouse Hippocampus*, Brain Structure and Function, 212(5), 403-416.
(<http://www.springerlink.com/content/r113706463v35457/fulltext.pdf>)
- [18] Harding, Antony J.; Halliday, Glenda M.; Kril, Jillian J. (1998). *Variation in Hippocampal Neuron Number with Age and Brain Volume*, Cerebral Cortex, 8(8), 710-718. (<http://cercor.oxfordjournals.org/cgi/reprint/8/8/710.pdf>)
- [19] Levy, William B (2004). Personal communication.
- [20] Hocking, Ashlie B. and Levy, William B (2006). *Gamma Oscillations in a Minimal CA3 Model*. Neurocomputing, 69(10-12), 1244-1248.
(<http://dx.doi.org/10.1016/j.neucom.2005.12.085>)
- [21] Hocking, Ashlie B. and Levy, William B (2007). *Theta-Modulated Input Reduces Intrinsic Gamma Oscillations in a Hippocampal Model*. Neurocomputing, 70(10-12), 2074-2078. (<http://dx.doi.org/10.1016/j.neucom.2006.10.086>)
- [22] Jin, Yaochu (2005). *A Comprehensive Survey of Fitness Approximation in Evolutionary Computation*. Soft Computing, 9(1), 3-12.
(<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.69.8393&rep=rep1&type=pdf>)

[23] Fast, Ethan; Le Goues, Claire; Forrest, Stephanie; Weimer, Westley (2009).

Designing Better Fitness Functions for Automated Program Repair. Genetic and Evolutionary Computing Conference (GECCO): 965-972.

(<http://www.cs.virginia.edu/~weimer/p/weimer-gecco2010.pdf>)

Appendix

Alternate random seeds for trace conditioning example

The following figures show the results for the same parameters used to generate Figure 17, but with random seeds 2 through 7.

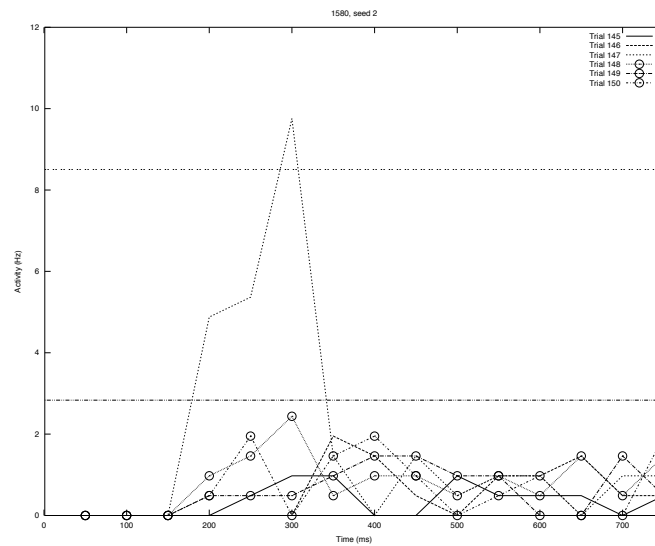


Figure 25. Performance for last few trials using random seed 2.

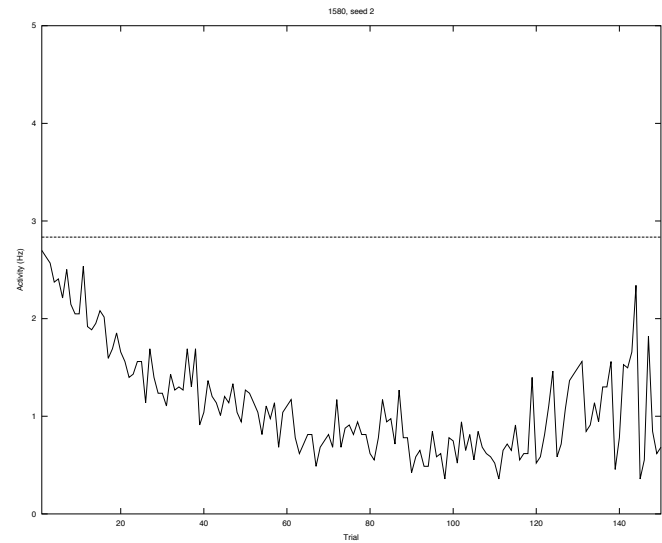


Figure 26. Across trial activity using random seed 2.

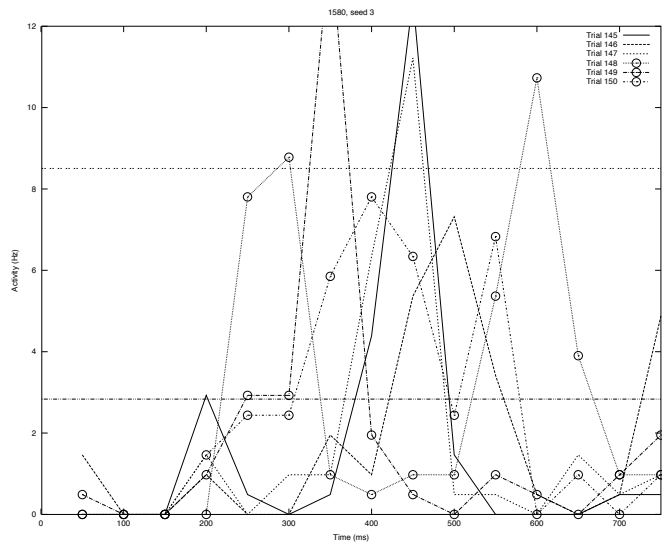


Figure 27. Performance for last few trials using random seed 3.

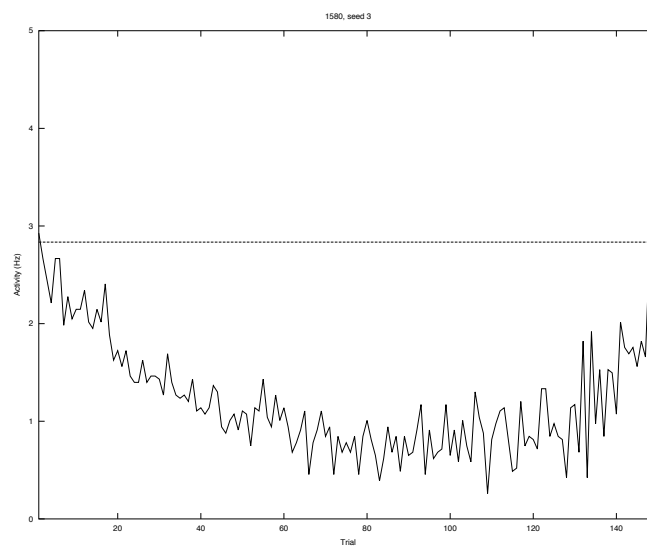


Figure 28. Across trial activity using random seed 3.

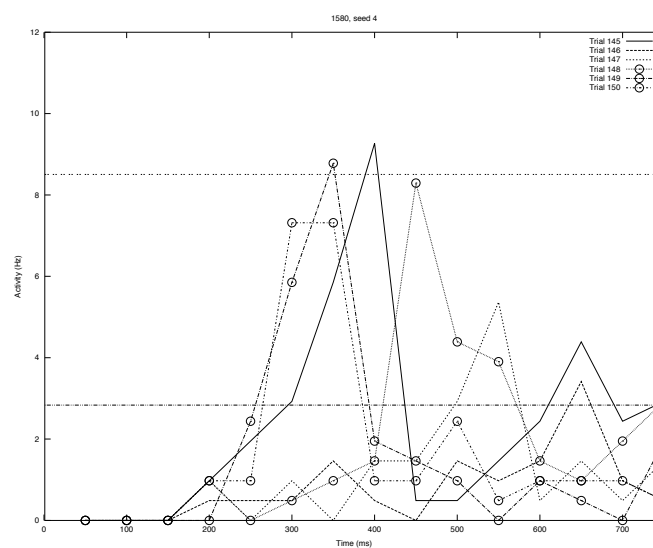


Figure 29. Performance for last few trials using random seed 4.

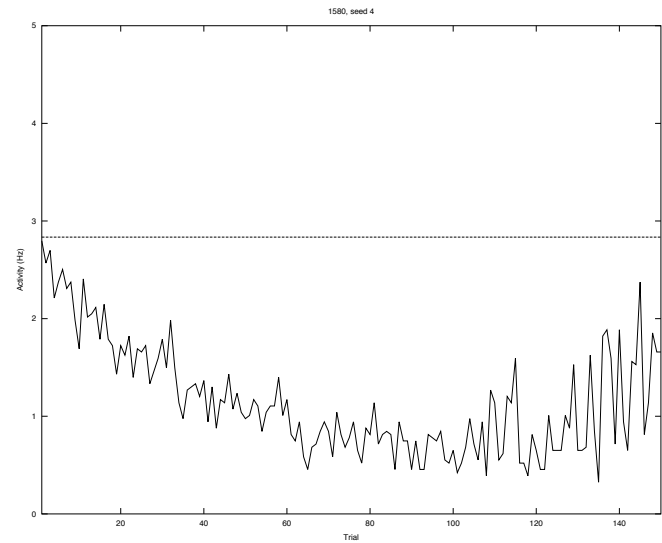


Figure 30. Across trial activity using random seed 4.

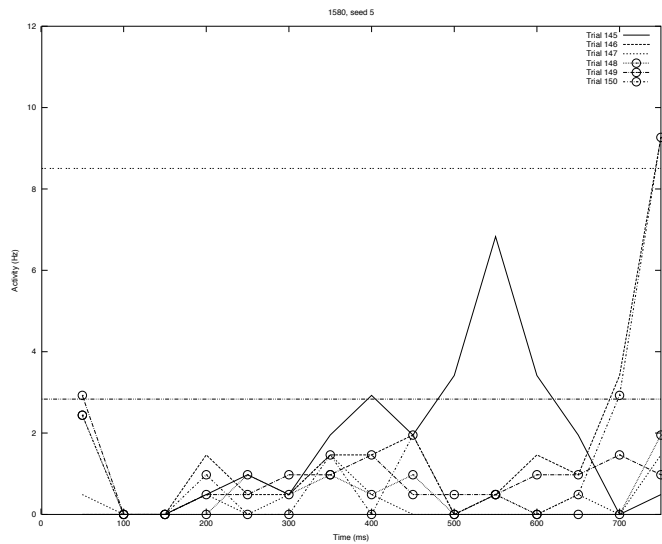


Figure 31. Performance for last few trials using random seed 5.

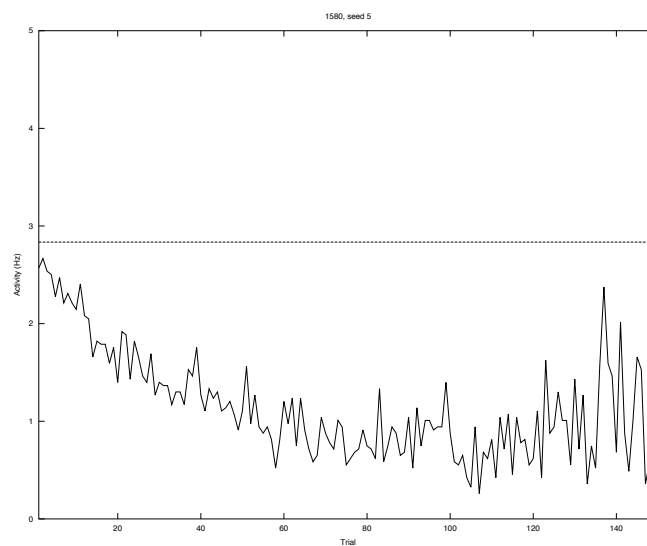


Figure 32. Across trial activity using random seed 5.

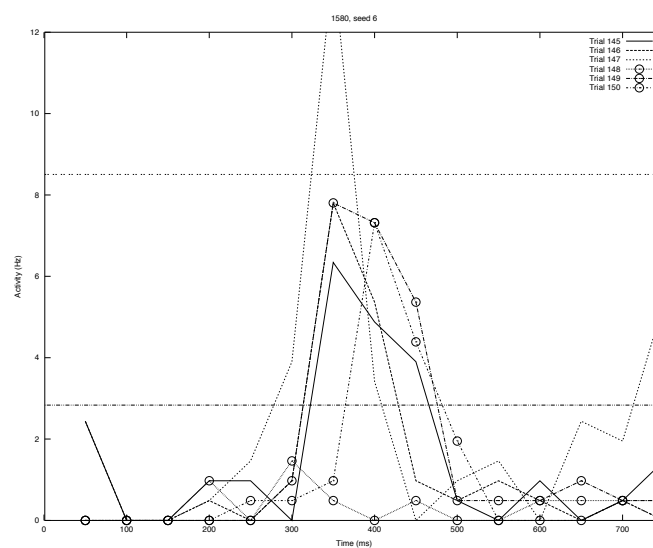


Figure 33. Performance for last few trials using random seed 6.

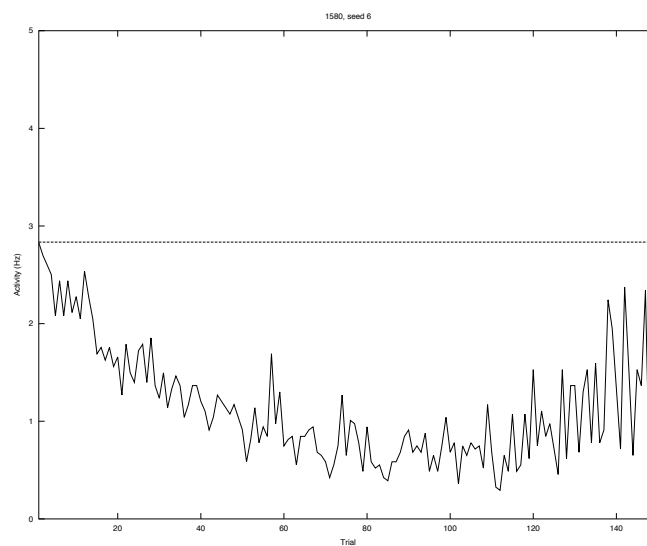


Figure 34. Across trial activity using random seed 6.

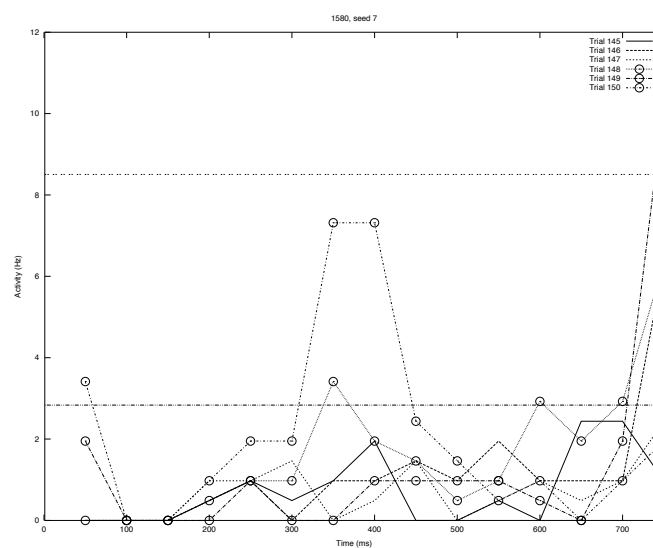


Figure 35. Performance for last few trials using random seed 7.

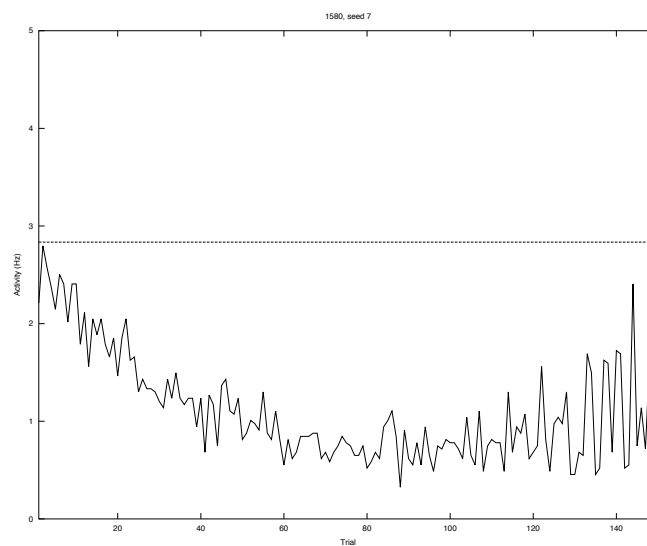
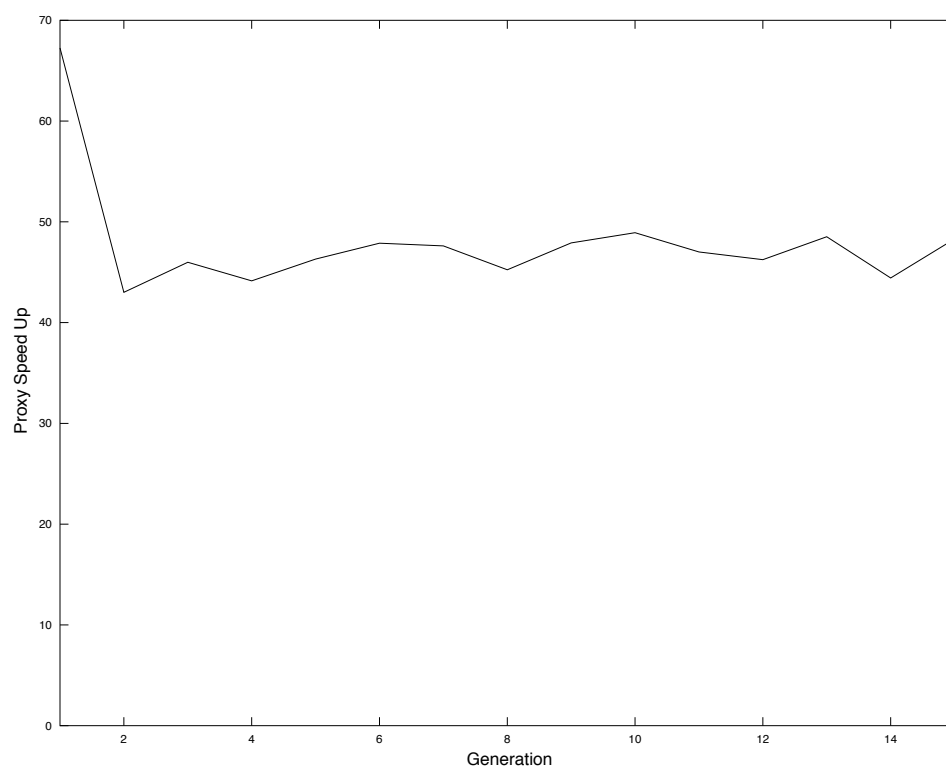


Figure 36. Across trial activity using random seed 7.

All generations of proxy computational speed up



Sample trace conditioning script

```

@CreateVar(
#  numTrainTrials 300

    numTrainTrials 150

    dwellTime 144          # ms

    OffRate 168.69374021197876 # ms

    OnRate 10.527192211200115          # ms

    ActivityHz 2.8347775148279837

    mePct 0.20021843519101787 # Percent of Activity due to me

    PyrToInternrnWtAdjDecayRate 10 # ms

    InternrnExcDecayRate 3.0510244761578473 # ms

    InternrnAxonalDelay 3 #ms

    filePostfix InsertBHere

    mySFR 0.18825035630935036

    KFBBBase 0.11863330057732645

    KFFBase 0.016282657407230942

    K0Base 0.02561081172035472

    TraceDuration 500

    setMu 0.015414817635324376

    setLambda 1.394322282916058

    minAxDelay 1

    maxAxDelay 4

    dendFilterWidth 8

    randomSeed 1580

)

@SetVar(deltaT 1)

```

```

@SetVar(Period ^Calc(round[^(dwellTime)/^(deltaT)]))

@PrintVar(deltaT)

#Default values

?If(strcmp[^(InternrnExcDecayRate),Insert:A:Here]=0) {
    @SetVar(InternrnExcDecayRate 3.5) # ms
}

?If(strcmp[^(filePostfix),Insert:B:Here]=0) {
    @SetVar(filePostfix tstActWithin.dat)
}

?If(strcmp[^(InternrnAxonalDelay),Insert:C:Here]=0) {
    @SetVar(InternrnAxonalDelay 2) # ms
}

?If(strcmp[^(dendFilterWidth),Insert:D:Here]=0) {
    @SetVar(dendFilterWidth 7) # ms
}

?If(strcmp[^(minAxDelay),Insert:E:Here]=0) {
    @SetVar(minAxDelay ^Num2Int(^Calc(round[2/^(deltaT)]))) # ms
}

?If(strcmp[^(maxAxDelay),Insert:F:Here]=0) {
    @SetVar(maxAxDelay ^Num2Int(^Calc(round[2/^(deltaT)]))) # ms
}

?If(strcmp[^(ActivityHz),Insert:G:Here]=0) {
    @SetVar(ActivityHz 1)
}

?If(strcmp[^(mySFR),Insert:H:Here]=0) {
    @SetVar(mySFR 0)
}

```



```

}
?If(strcmp[^(OffRate),Insert:I:Here]=0) {
    @SetVar(OffRate 100)
}
?If(strcmp[^(OnRate),Insert:J:Here]=0) {
    @SetVar(OnRate 15)
}
?If(strcmp[^(KFBBBase),Insert:K:Here]=0) {
    @SetVar(KFBBBase 0.01903)
}
?If(strcmp[^(KFFBase),Insert:L:Here]=0) {
    @SetVar(KFFBase 0.00207)
}
?If(strcmp[^(K0Base),Insert:M:Here]=0) {
    @SetVar(K0Base 0.02726)
}
?If(strcmp[^(TraceDuration),Insert:N:Here]=0) {
    @SetVar(TraceDuration 500) # ms
}
?If(strcmp[^(setMu),Insert:O:Here]=0) {
    @SetVar(setMu 0.01) # ms
}
?If(strcmp[^(setLambda),Insert:P:Here]=0) {
    @SetVar(setLambda 0.5) # ms
}
?If(strcmp[^(mePct),Insert:X:Here]=0) {
    @SetVar(mePct 0.3) # ms
}

```

```

@PrintVar(InternrnExcDecayRate filePostfix InternrnAxonalDelay)

@CreateVar(halfWidth ^Calc(^dendFilterWidth)/2))

@CopyData(-to DTSFiltA -from "^Fn(^deltaT,^(halfWidth),1-exp[-
^(halfWidth)/5*^(tFn)])" -type mat -T)
@CopyData(-to DTSFiltB -from
"^Fn(^halfWidth+^(deltaT),^(dendFilterWidth),exp[-
^(halfWidth)/5*[(tFn)-^(halfWidth)])]" -type mat -T)
@AppendData(-from 2 DTSFiltA DTSFiltB -to DTSFiltT -type mat)
@CopyData(-to DTSFilt -from DTSFiltT -type mat -T)
@CopyData(-to WtFilt -from "^Fn(^deltaT,1,1)" -type mat)

@SetVar(
    title "Trace Conditioning"
    mu ^(setMu)
    Phase 0
    Activity ^Calc(^ActivityHz) * ^deltaT / 1000)
    ni 2048
    DendriteToSomaFilter DTSFilt
    SynapseFilter WtFilt
    FBInternrnAxonalDelay
    ^Num2Int(^Calc(round[^(InternrnAxonalDelay)/^(deltaT)]))
    FFInternrnAxonalDelay 1 # to allow it to be quasi in-sync
    InternrnExcDecay ^Calc(1-exp[-^(deltaT)/^(InternrnExcDecayRate)])
    ResetPattern zeros
)

```

```

@MakeRandSequence(-name zeros -len 1 -p 0)

@SetVar(
    ExtExc ^Calc(^ (ni) * ^ (Activity))
)
@MakeSequence(-name InitPtn -len 1 -non 1 -Nstart 1)

?If(exists[wNoise]) { @SetVar(wNoise ^ (mySFR)) }
?If(exists[SynFailRate]) { @SetVar(SynFailRate ^ (mySFR)) }

@CreateVar(
    # mePct varies from 0.2 to 0.5
    me ^Calc(round[^ (ni) * ^ (mePct) / 10]) # 10 is arbitrary
)
#@SaveData(-from me -to me.dat)
#@SaveData(-from mePct -to mePct.dat)
#@SaveData(-from ActivityHz -to ActivityHz.dat)

?If(exists[PyrToInternrnWtAdjDecay]) {
    @SetVar(PyrToInternrnWtAdjDecay ^Calc(exp[-
    ^ (deltaT) / ^ (PyrToInternrnWtAdjDecayRate)]))
}
?If(exists[KdAdjDecay]) {
    @SetVar(KdAdjDecay ^Calc(exp[-
    ^ (deltaT) / ^ (PyrToInternrnWtAdjDecayRate)]))
}

```

```

@SetVar(

  NMDArise ^Calc(ceil[^(OnRate) / ^(deltaT)])

  theta 0.5

  Con 0.1

  KFF ^Calc(^(KFFBase) * ^(dendFilterWidth))

  KFB ^Calc(^(KFBBase) * ^(dendFilterWidth))

  K0 ^Calc(^(K0Base) * ^(dendFilterWidth))

  wStart ^Calc(^(ActivityHz) * ^(OffRate) * ^Calc(1-^(mySFR)) /
1000)

  Reset 0 # Circular sequence

  alpha ^Calc(exp[-^(deltaT)/^(OffRate)])

  MidPoint ^Calc(^(mePct) * ^(ni) * ^(Activity) / ^(me))

  xNoise ^Calc(^(mePct) * ^(ni) * ^(Activity) / ^(me))

  xTestingNoise ^Calc(^(mePct) * ^(ni) * ^(Activity) / ^(me))

  lambdaFB ^(setLambda)

## Begin Izhikevich Block ##

# Remove for Classical model #

  IzhA 0.02437434474636943

  IzhB -0.09098031934366621

  IzhC -56.084834380015565

  IzhD 5.715417424859181

  IzhvStart -64.12632551580455 # stable point

  IzhvStart ^Calc(-0.09098031934366621 * -64.12632551580455) #

stable point -60 * -0.1

  IzhIMult 11.361992401899078

### End Izhikevich Block ###

)

```

```

@MakeRandSequence(-name InitPtn -len ^Calc(100/^(deltaT)) -p
^Calc(^ (Activity)/^(MidPoint)))

@CreateVar(
    startN 1
    lastN ^Calc(2 * ^(me))
    tonelen ^Num2Int(^Calc(150 / ^(deltaT)))
    pufflen ^Num2Int(^Calc(100 / ^(deltaT)))
    tracelen ^Num2Int(^Calc(^ (TraceDuration) / ^(deltaT)))
)

?If(strcmp[^(randomSeed),Insert:Seed:Here]=0) {
    @SetVar(randomSeed 1)
}

@SetVar(seed ^(randomSeed))

@SeedRNG()

@CreateNetwork(-mindelay ^(minAxDelay) -maxdelay ^(maxAxDelay) -dist
uniform -low ^Calc(0.9 * ^(wStart)) -high ^Calc(1.1 * ^(wStart)))

@DeleteData(InitPtn)

@MakeRandSequence(-name InitPtn -Nend ^(ni) -len ^(Period) -p
^Calc(^ (mePct)*^(Activity)/^(MidPoint)))

@MakeSequence(-name blank -len ^(tonelen) -non 0 -Nstart 1)

@MakeSequence(-name tone -len ^(tonelen) -st ^(tonelen) -non ^(me))
@MakeSequence(-name trace -len ^(tracelen) -st ^(tracelen) -non 0)

```

```

@MakeSequence(-name puff -len ^(pufflen) -st ^(pufflen) -non ^(me) -
Nstart ^Calc(^(me)+1))

@MakeSequence(-name nopuff -len ^(pufflen) -st ^(pufflen) -non 0)

@AppendData(-to trainTraceSeq -from 3 tone trace puff)

@AppendData(-to testTraceSeq -from 3 tone trace nopuff)

@CreateVar(firstRecur ^Calc(^(lastN)+1))

@PrintVar(deltaT alpha firstRecur Activity me MidPoint)


@SetVar(seed ^(randomSeed))

@SeedRNG()


@PrintVar(deltaT alpha Activity me MidPoint)


@Test(-name testTraceSeq -time ^SequenceLength(-from testTraceSeq) -
nocomp -norecord 7 TestingThresholds TestingBusLines
TestingIntBusLines TestingKWeights TestingInhibitions
TestingFBInternrnExc TestingFFInternrnExc)

@PrintVar(AveTestAct)

@CreateVar(tempTest 0)

@CreateVar(tempData 0)

%(i 1 ^(numTrainTrials)) {

    @ResetFiring()

    @DeleteData(InitPtn)

    @MakeRandSequence(-name InitPtn -Nend ^(ni) -len ^(Period) -p
^Calc(^(mePct)*^(Activity)/^(MidPoint)))

    @Test(-name InitPtn -time ^SequenceLength(-from InitPtn) -nocomp -
norecord 7 TestingThresholds TestingBusLines TestingIntBusLines

```

```

TestingKWeights TestingInhibitions TestingFBInternrnExc
TestingFFInternrnExc)

    @DeleteData(trace)

    @MakeRandSequence(-name trace -Nend ^(ni) -len ^(tracelen) -p
^Calc(^ (mePct)*^(Activity)/^(MidPoint)))

    @AppendData(-to trainTraceSeq -from 3 tone trace puff)

    @MakeSequence(-name nopuff -len ^(pufflen) -st ^(pufflen) -non 0)

    @AppendData(-to testTraceSeq -from 3 tone trace nopuff)

    @Train(-name trainTraceSeq -trials 1 -nocomp -norecord 7
TrainingThresholds TrainingBusLines TrainingIntBusLines
TrainingKWeights TrainingInhibitions TrainingFBInternrnExc
TrainingFFInternrnExc)

    @PrintVar(AveTrainAct)

    ?If(^ (i) = 1) {

        @FileReset(fit_trn_mean_act.dat fit_trn_ssd_act.dat)

        %(j 1 701 50) {

            # This is a little confusing because we're using N (neuron)
where P (timestep) makes more sense

            # That's because CopyData was designed for 0/1 patterns and
not activities

            @CopyData(-from TrainingActivity -Nstart ^(j) -Nend ^Calc(^ (j)
+ 49) -to subAct -type mat)

            @SetVar(tempData ^Calc(mean[subAct]))

            @SetVar(tempData ^Calc(1000 * ^(tempData) / ^(deltaT))) #
Convert to Hz

            @SaveData(-from tempData -to fit_trn_mean_act.dat -append)

        }

        @SetVar(tempData ^Calc(sqrt[var[TrainingActivity]]))

```

```

    @SetVar(tempData ^Calc(1000 * ^(tempData) / ^(deltaT))) #
Convert to Hz

    @SaveData(-from tempData -to fit_trn_ssd_act.dat)

}

@DeleteData(TrainingBuffer)

@SetVar(tempTest 0)

?If(^i = 1) {

    @SetVar(tempTest 1)

}

# ?If(^i = ^(numTrainTrials)) {
#     @SetVar(tempTest 1)
# }

?If(^i % 50 = 0) {

    @SetVar(tempTest 1)

}

?If(^tempTest = 1) {

    @ResetFiring()

    @DeleteData(InitPtn)

    @MakeRandSequence(-name InitPtn -Nend ^(ni) -len ^(Period) -p
^Calc(^mePct)*^(Activity)/^(MidPoint)))

    @Test(-name InitPtn -time ^SequenceLength(-from InitPtn) -nocomp
-norecord 7 TestingThresholds TestingBusLines TestingIntBusLines
TestingKWeights TestingInhibitions TestingFBInternrnExc
TestingFFInternrnExc)

    @DeleteData(trace)

    @MakeRandSequence(-name trace -Nend ^(ni) -len ^(tracelen) -p
^Calc(^mePct)*^(Activity)/^(MidPoint)))

    @DeleteData(nopuff)

```



```

    @MakeRandSequence(-name nopuff -Nend ^(ni) -len ^(pufflen) -p
^Calc(^(mePct)*^(Activity)/^(MidPoint)))

    @AppendData(-to testTraceSeq -from 3 tone trace nopuff)

    @Test(-name testTraceSeq -time ^SequenceLength(-from
testTraceSeq) -nocomp -norecord 7 TestingThresholds TestingBusLines
TestingIntBusLines TestingKWeights TestingInhibitions
TestingFBInternrnExc TestingFFInternrnExc)

    @PrintVar(AveTestAct)

    ?If(^(i) = 1) {

        @FileReset(fit_tst_mean_act.dat fit_tst_ssd_act.dat)

        %(j 1 701 50) {

            # This is a little confusing because we're using N (neuron)
where P (timestep) makes more sense

            # That's because CopyData was designed for 0/1 patterns and
not activities

            @CopyData(-from TestingActivity -Nstart ^(j) -Nend
^Calc(^(j) + 49) -to subAct -type mat)

            @SetVar(tempData ^Calc(mean[subAct]))

            @SetVar(tempData ^Calc(1000 * ^(tempData) / ^(deltaT))) #
Convert to Hz

            @SaveData(-from tempData -to fit_tst_mean_act.dat -append)

        }

        @SetVar(tempData ^Calc(sqrt[var[TestingActivity]]))

        @SetVar(tempData ^Calc(1000 * ^(tempData) / ^(deltaT))) #
Convert to Hz

        @SaveData(-from tempData -to fit_tst_ssd_act.dat)

        @SaveData(-from tempTest -to fit_quick.ready)

    }

```

```

?If(^i) % 50 = 0) {
    @FileReset(fit2_:^(i):.dat)
    %(j 1 701 50) {
        @CopyData(-from TestingBuffer -Nstart ^Calc(^me) + 1) -Nend
^Calc(^me) * 2) -Pstart ^(j) -Pend ^Calc^(j) + 49) -to subBuff -
type mat)
        @SetVar(tempData ^Calc(mean[subBuff]))
        @SaveData(-from tempData -to fit2_:^(i):.dat -append)
    }
    @SaveData(-from tempTest -to fit2_:^(i):.dat.ready)
}
@DeleteData(TestingBuffer)
}
}
@DeleteData(TrainingActivity)

```