seit 1558

Learning delayed-response tasks in spiking neural networks

Diplomarbeit

zur Erlangung des akademischen Grades

Diplom-Informatiker

Friedrich-Schiller-Universität Jena

Fakultät für Mathematik und Informatik

eingereicht von Jan Huwald

geb. am 27.05.1985 in Gera

Betreuer: Prof. Dr. Jochen Triesch

Jena, 25.11.2011

Abstract

Working memory is the brain's ability to transiently store and process task related information. Biological experiments show that working memory tasks are influenced by the neurotransmitter dopamine which is thought to act as reward signal. Existing in-silico models reproduce working memory abilities but not their acquisition. I show that a working memory task can be learned using a dopamine reward. To this end a biologically plausible recurrent spiking neural network with leaky integrate-and-fire neurons, dopamine-modulated spike-timing dependent plasticity and intrinsic plasticity is trained on a delayed response task. A small but statistically significant performance increase above chance is demonstrated.

Contents

1 Introduction

Goal-directed action in complex environments requires that our brain is able to transiently store task-relevant information: The integration of temporally scattered cues, acting on past events and manipulation of mental objects are unthinkable otherwise. This ability is captured by the concept of working memory (WM) and has been subject of intense research over the past five decades [Bad03]. Models of WM exist across subjects and scales. Before diving into the depths of the computational neuroscientific aspect illuminated in this thesis, let us visit selected lighthouses of this wide research effort.

George Miller—a co-inventor of the term working memory—became famous for two psychometric observations connected by the number seven. First, that after a single presentation untrained humans can remember approximately seven different entities[1] for a short duration, relatively independent of the set from which those entities are drawn (digits, letters, words) and the precise circumstances of the presentation. And second, that the cognitive channel capacity of participants classifying one-dimensional stimuli on an absolute scale is approximately $\log_2 7 \approx 2.5$ bits per judgment—no matter if stimuli was visual, auditory or gustatory [Mil56].

Initially these findings were integrated into the modal model of memory. It divides information storage into sensory, primary (short term, including WM) and secondary (long term) storage [HM96]. But its prediction of a single, unitary working memory was disproved by Baddeley and Hitch when they discovered that WM intensive tasks can be executed while the participant has to hold up to eight digits—without a major performance disruptions in one of both tasks [BH74]. This observed independence led to the formulation of the multi-compartment model of working memory. It consists of the attention controlling central executive, the phonological loop, the visuo-spatial sketchpad and the episodic buffer [Bad10, Bad00]. Interestingly Baddeley describes the phonological loop as a kind of attractor: To refresh "fading memory traces" a short auditory stimulus is constantly self-rehearsed in real time [Bad03]. This motif of operation recurs in neural working memory models. Because unrefreshed memory is assumed to have a limited time span of existence the working memory is limited: the rehearsal of all elements to remember must not exceed this span, or elements will not be refreshed in time. This yields a possible explaination for Millers seven [Bad03].

But this number can also be explained by neural scale models [LI95]. In accordance to the multi-compartmental hypothesis of WM, neural activity related to WM has been located to different regions of the cortex, depending on the task at hand [CUKH96, UCH98, Fus73]. Visuo-spatial working memory for example has been shown to extensively depend on neurons in the dorsolateral prefrontal cortex by primate physiological studies [Fus73] and brain imaging [CUKH96]. Locating WM

[1] In the WM literature these entities are usually called chunks of information to underline their missing grounding in information-theoretic concepts.

relevant regions of the brain to this precision allows recording the activity of single task relevant neurons. The tenor of several studies is that neural activity in those regions is elevated during the entire task—starting from the perception of the information until its obsolescence after it has been retrieved [RM02, Fus73, HT09]. Disturbing this elevated activity, drastically reduces the task performance [FBGR89]. The precise time courses of observed neural activity differ widely across studies.

Computational models of working memory can improve the situation by offering a quantitative testbed for hypothesis. Coarse grained WM models successfully implemented parts of the multi-compartment model of WM with mathematical rigor and quantitatively tested its predictions against psychometrical observations [BH99].

Finer grained models use biologically somewhat plausible neurons to map WM abilities to neural system dynamics. A common choice, also used in this work, are spiking neural networks. The necessary concepts of leaky integrate-and-fire neurons, spike-timing dependent plasticity (STDP) and its dopamine modulation as well as intrinsic plasticity (IP) are introduced in the methods section 2.1 together with their mathematical formulation within the developed model.

Restricting to spiking neural networks, all computational WM models[2] have in common that they implement information storage by selecting among multistable states. They can be distinguished by the scale at which they locate those: at synapse, neuron or network level [DSS00, MBT08].

Mongillo, Barak, and Tsodyks, 2008 suggest that the presynaptical residual calcium concentration is used for information storage[3]: An external cue increases the spike frequency of a corresponding subpopulation of a recurrent neural network and thereby increases the presynaptical calcium concentration of intrapopulation synapses. Later, a readout signal—a small external drive given to the entire network—reactivates the subpopulation with the highest calcium concentration, reproducing the last given input [MBT08].

A neuron scale bistability has been proposed, based on an inverted bell-shaped voltage dependency of the synaptic efficacy. In presence of two external oscillations of high frequency difference, neurons with two stable states (low- and high-frequent) emerge [LI95]. Interestingly the resulting network is the only of those presented here, which allows to hold novel stimuli [DSS00].

This thesis' model resides on the third multistability scale: network scale explanations of information storage, in which a subset of neurons exhibits changed firing rates to hold the stimuli. There are two ideas of how to maintain this state [DSS00]. The first may be tracked back until Hebb, who proposes that a subset of the network features a cluster of neurons with strongly excitatory, reciprocal connections [Heb49], whereas intercluster connections are dominated by inhibition. Once most neurons of one set are activated by an external stimuli, this recurrent topology causes enough input current to maintain elevated firing rates, even after the external input disappears, while at the same time preventing competing cell assembling from becoming active [BW01, CBGRW00].

[2] known to the author

[3] Although this model is used as example for storage based on a synapse scale bistability, it has to be noted that the negative feedback between different bistable units (synapses with high or low calcium concentration) is mediated by network scale inhibition.

The second—synfire chains—is a slight variation: instead of maintaining elevated firing activity within a homogeneous group of reciprocally connected neurons, the population of active neurons is further divided into subgroups connected by feed-forward connections. One subgroup activates the next, with persistent activity occurring if the last subgroup is connected to the first [DGA99]. Noteworthy for working memory tasks is the property of synfire chains in a network with variable conductance delays, that far more synfire chains may exist then neurons [Izh06].

The model developed in this thesis is agnostic to whether the information is stored in synfire chains or recurrent excitation within homogeneous cell assemblies. Due to the simplicity of the used leaky integrate-and-fire neuron, information storage based on neuron or synapse scale is ruled out, however.

Learning working memory The models considered so far have been static, mostly hand-constructed to perform an WM task. But working memory capabilities are not given by birth. They have to be acquired over a long time span: Psychometrical experiments show that WM performance of children increases between age 4 and 14, independent of the task [GPAW04]. During the childhood several qualitative jumps occur, during which new interconnections between different working memory subsystems emerges [LN98, GH93].

On a much smaller time scale, the performance is increased if the entity to reproduce has been encountered before [SB98]. Differences between familiar and novel stimuli have been tracked down to neural scale: Novel stimuli produce higher activities during parts of the WM task [BGB+98, RM02].

These observations lead to the hypothesis that working memory can be learned. The question remains how.

Dopamine—a neurotransmitter—is known to play a crucial role for learning [MHC04, DR09] and working memory [WKH97, PAF04, GMWK07]. Dopaminergic neurons project into several WM-relevant areas of the brain, among them the aforementioned dorsolateral prefrontal cortex [WKH97]. Several cues and hypothesis for the role dopamine plays for working memory exist. It is known that the elevated dopamine concentrations at the D1 receptor can induce persistent firing without recurrent excitation [SLF+09]. This led to the hypothesis that dopamine is a switch for a neuron scale bistability which implements WM [DH11].

But dopamine is also widely acknowledged as carrier of a reward or reward-predictor signal [MHC04, GMWK07]. For delayed response tasks this function is established solid enough to predict a subjects accuracy by measuring its dopamine release [PAF04]. Dopamine is known to modulate Hebbian learning [MHC04]: increased dopamine levels at D1/D5-receptors enhances long term potentiation (LTP) and long term depression (LTD) [OL96, LMV06]. The focus of this work lies on the role of dopamine as reward signal.

One computational model based on these observations is that of dopamine-modulated spike-timing dependent plasticity (STDP) [Izh07]. It has been shown analytically to be able to learn to classify spatial and temporal firing patterns [LPM08], even if action and reward are temporally distant [Izh07]. The model is further introduced in section 2.1.3.

Psychological computational models have been shown to learn working memory using a dopamine reward signal [TNC09]. For biologically plausible neural scale models, it is an open question if working memory can be learned using such a dopamine delivered reward signal.

Contribution and overview In this work a biologically plausible model network of spiking neurons, governed by dopamine-modulated spike-timing dependent plasticity is developed. It is shown that this model can learn delayed response tasks, given a dopamine reward.

In section 2.1 the model is introduced and motivated in the light of experimental and theoretical results. After defining the delayed response task to learn (section 2.2) a novel method for analyzing propagation of causal influence is introduced in section 2.3. The model is analyzed using the state trajectory of individual models variables, the global weight development (section 3.1) and the proposed causality tracking (section 3.2). To test generality and succinctness, the model's performance is measured under parameter variation (section 3.3) and component omission (section 3.4). The results are compressed and discussed in chapter 4. The software developed for this work is presented in Appendix 8.2 to 8.5.

2 Methods

2.1 Model

The neural network model used in this work incorporates a range of dynamics of different time scales, concepts, and authors. On a formal level it can be split into almost orthogonal subsystems which share only a few interfacing state variables. Along these lines the following subsections will define the model, consisting of the neuron model, the plasticity mechanism, its reward modulation, the homeostatic mechanism, and the network topology.

The model is a pure hybrid differential delay equation system: *Differential*, because variables depend on their deviation; *hybrid*, because the differential equations are supplemented with well defined jump discontinuities; *delay*, because variables depend on their past value; and *pure*, because it does not deviate from any of the conditions above in any case.

In addition to this characterization the model also is constructed towards biological plausibility. Beyond choosing sub-models and parameters that are experimentally justified, this requires filling the gaps (where no experimental evidence exists) obeying the metaphysical constraints locality, causality, and finiteness: *causality* requires that a no variable depends on the future (or the present, in a way that creates circular dependencies); *finiteness* demands finite bounds on all variables; *locality* requires that a variable only depends on variables at the same site or its superstructure.

The hybrid nature of the model implies, that to define the dynamic of a variable x one needs to define a differential term $\frac{\mathrm{d}}{\mathrm{d}\,t}x$ and a difference term Δx. The nonzero positions of the difference function yield the jump discontinuities $D = \{t : \Delta x \neq 0\}$, of which at most countable many may exist. The time course of x can then be defined by iterating over $D = \{t_0 < t_1 < \dots\}$. For $\tilde{t} \in (t_i, t_{i+1})$ one only needs the differential function:

$$x(\tilde{t}) = x(t_i) + \int_{t_i}^{\tilde{t}} (\frac{\mathrm{d}}{\mathrm{d}\,t}x)\,\mathrm{d}\,t.$$

For the value at the next discontinuity t_{i+1} one also needs the difference function:

$$x(t_{i+1}) = x(t_i) + \Delta x(t_{i+1}) + \lim_{\tilde{t}\to t_{i+1}-} \int_{t_i}^{\tilde{t}} (\frac{\mathrm{d}}{\mathrm{d}\,t}x)\,\mathrm{d}\,t.$$

In the rest of this section this variable naming scheme and evaluation strategy will be assumed without further notice.

The model is implemented in the domain-specific language of the RaSimu simulator, which was developed during this work. Beyond representing elements of $\mathbb{R}$ as

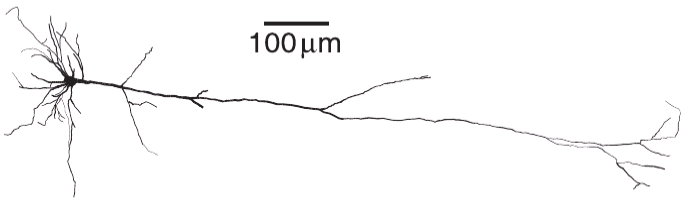


Figure 2.1: Drawing of a biological neuron, copied from [RPPD05]. Spikes travel from left to right: Dendrites collect spikes emitted from other neurons via synapses, which evoke postsynaptical electrical potentials. If the spatiotemporal sum of these potentials exceeds the firing threshold at the soma, the neuron emits a spike. After traveling the axon the delayed spike reaches its target neurons.

floating point numbers, the simulator exactly recapitulates the dynamic of the formal model. This in turn requires that every differential term in the formalism above has an analytical integral. For the model implementation, see Appendix section 8.3; For details on the simulator section 8.4.

2.1.1 Leaky integrate-and-fire neurons

During the short history of the field of computational neuroscience a remarkable amount of qualitatively different models of neurons have been developed [BRC+07]: their smallest subunits range from synapse fine structure to neuron populations. Contemporary models may be distinguished on the way they represent the electrical stimulation of the neuron membrane: rate based, spike based or conductance based[1] (e.g. Hodgkin-Huxley) [BRC+07].

Conductance based models yield the highest level of detail and agreement with biological experiments [BRC+07, MB01, pp 34-44]. They spatially model neurons by dividing their surface into patches and solve biophysical partial differential equations simulating the electrical process of the neuron membrane [MB01, pp 34-44]. The obvious computational complexity requirements and the sheer size of the associated parameter space render conductance based models inadequate for the qualitative questions addressed in this work.

Rate based models have dominated the research for most of the time [MB01, pp 6-7], leading to popular abstractions like the perceptron and other fundamentals of early artificial intelligence. Rate refers to an analog quantity, resembling the frequency of electrical spikes between two neurons (or populations, or experiments) [MB01, pp 7-11] and is the only quantity of information transfer in such models.

Experimental evidence of the recent years suggest that the rate based approach is too simple for describing brain activity [BRC+07, MB01, p 7]. Instead the precise spike timing [MS95], spike phase [Nad09, CK09], and interspike interval [MMS+09] seems crucial for the qualitative network behavior[2]. These observations reinforced

[1] With conductance based I refer to compartmentalized variants of these models, though it should be noted that conductance based models without spatial structure exist, too.

[2] These citations are only a few examples and by no means the first experiments suggesting the

the use of spiking neural network models, whose level of detail is intermediate: the spike rate (the single output of a neuron and input of a synapse) is replaced with the superposition of time shifted action potentials with well defined shape [MB01, pp 5]; the temporal behavior of conductance based models is abstracted with a synapse specific transmission delay.

Our model consists of N neurons of which each has a membrane potential u_i (with $i, j \in 1, N$ for the rest of this text). The potential exponentially decays towards a resting potential u_{rest}. I however set $u_{rest} = 0$ to simplify the model description[3]:

$$\frac{\mathrm{d}}{\mathrm{d}\,t} u_i = -u_j/\tau_u. \tag{2.1}$$

The aforementioned time of an (outgoing) action potential is defined as the time when the membrane potential reaches the firing threshold: $u_i \geq u_{\text{thresh}}$. A biological action potential has a nonzero duration with a temporal fine structure containing a sharply rising and falling phase followed by an undershot. In my model I reduce this shape to a Dirac δ function[4]. When a spike is emitted then u_i is reset to the resting potential immediately. A neurons output behavior can thus be characterized by a set of spike times T_i defined as:

$$T_i := \{t \in \mathbb{R} : u_i(t) = 0 \wedge \lim_{t' \to t-} u_i(t') > 0\}. \tag{2.2}$$

Each neuron i has S_i synapses with two important properties: synaptic weight $w_{ij}(t)$ and transmission delay τ_{ij}. A spike originating at time t from neuron i arrives at neuron j at time $t + \tau_{ij}$ and is multiplied by w_{ij} before being applied to u_j. Thus the set T_{ij} of times of spikes arriving at neuron j over synapse ij is:

$$T_{ij} = \{t : t - \tau_{ij} \in T_i\} \tag{2.3}$$

This set of incoming spike times is also the set of all jump discontinuities of u_i whose difference term can now be fully defined. The case denoted with (*) leads to the creation of a new spike.

$$\begin{aligned} u_{i,\text{input}}(t) &:= \sum_{j:t\in T_{ij}} w_{ij}(t) \\ u_{i,\text{pre}}(t) &:= \lim_{t' \to t-} u_i(t') \\ u_{i,\text{post}}(t) &:= u_{i,pre}(t) + u_{i,\text{input}}(t) \\ \Delta u_i(t) &= \begin{cases} u_{i,\text{input}}(t) & \text{if } u_{i,\text{post}}(t) < u_{\text{thresh}} \\ -u_{i,\text{pre}}(t) & \text{if } u_{i,\text{post}}(t) \geq u_{\text{thresh}} \quad (*) \end{cases}. \end{aligned} \tag{2.4}$$

The model described so far is called a leaky integrate-and-fire neuron [BRC+07, MB01]. It integrates the current incoming from its synapses (2.5), looses its mem-

importance of the single spike.

[3]This does not reduce biological plausibility as the firing threshold is shifted by the same amount and the system is invariant under this linear transformation

[4]By definition $\delta(t) = 0$ for $t \neq 0$ and $\int_{-\infty}^{\infty} \delta \,\mathrm{d}\, t = 1$.

brane potential by firing an action potential if a threshold is reached (2.3) or leak it over time otherwise. It is equivalent to the simple spiking neuron model described in [MB01, pp 17-20] (with a zero refractory kernel η_i) and compatible with the spike based models described in [BRC+07].

The model described so far has no refractory term beyond resetting u_i to u_{rest}. This led to frequent resonance disasters on millisecond timescales through positive feedback. To prevent this, I add an absolute refractory period. If a neuron fires at t then u_i is not only reset to zero but forced to remain zero for $\tau_{\text{refractory}}$ seconds. In this time no incoming spike has any effect on the neuron. Thus 2.5 becomes:

$$\Delta u_i(t) = \begin{cases} u_{i,\text{input}}(t) & \text{if } u_{i,\text{post}}(t) < u_{\text{thresh}} \wedge T_i \cap [t - \tau_{\text{refractory}}, t) = \emptyset \\ -u_{i,\text{pre}}(t) & \text{if } u_{i,\text{post}}(t) \geq u_{\text{thresh}} \wedge T_i \cap [t - \tau_{\text{refractory}}, t) = \emptyset \end{cases}. \tag{2.5}$$

The absolute refractory period can be viewed as a crude approximation of the "soft" refractory period of biological neurons.

2.1.2 Spike-timing dependent plasticity

The network of leaky integrate-and-fire neurons described so far has constant weights. To be able to learn synaptic plasticity—any mechanism to translate network activity into persistent weight changes—is required. In our model spike-timing dependent plasticity (STDP) is used for this purpose. It is a temporally asymmetric form of Hebbian learning [SG10]. Despite open questions STDP is considered a biologically plausible phenomenological description of its experimental correlates: long-term potentiation (LTP) and long-term depression (LTD) [SG10, Sho07].

The basic principle of STDP is depicted in Figure 2.2. The weight change of a given synapse is a function of the time difference between a pre- and a postsynaptic spike [SG10]. If the presynaptic spike precedes the postsynaptic potentiation occurs, otherwise depression is the effect [SG10]. Note that this mechanism strengthens a connection if a presynaptic excitatory neuron's spike is a cause for the postsynaptic neuron to fire. This is unison with Hebbs intention behind his famous "fire together, wire together" rule [MDG08].

A simple mathematical formulation of the STDP principle might calculate the weight change w_{ij} induced by a spike pair occurring at t_j^{pre} (presynaptic) and t_i^{post} (postsynaptic) as

$$\Delta w_{ij} = W(t_i^{\text{post}} - t_j^{\text{pre}}) = W(\Delta t) := \begin{cases} +A_+ \exp(\Delta t/\tau_+) & \text{if } t_i^{\text{post}} < t_j^{\text{pre}} \\ -A_- \exp(\Delta t/\tau_-) & \text{if } t_i^{\text{post}} > t_j^{\text{pre}} \end{cases} \tag{2.6}$$

with $\Delta t = |t_i - t_j|$. W is denoted the learning window, A_+ and A_- are learning rates and τ_+, τ_- determine the learning window's length. The different parameters for the potentiation and depression case are required to match experimental correlates [MDG08] and because of theoretical analysis on stability and long-term development [MDG08, LPM08].

Obviously this description is ambiguously and problematic in several ways. Because the variety of phenomenologically different STDP formulations appears along these ambiguities we will dissect them in detail below.

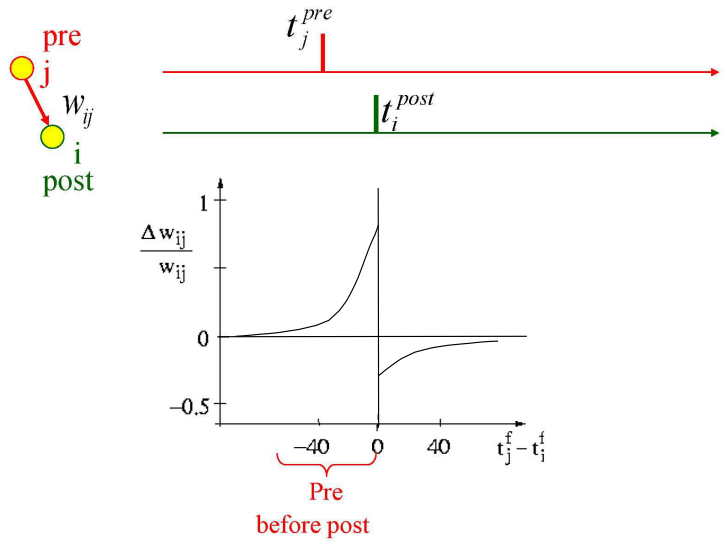


Figure 2.2: The weight change of a synapse induced by a pair of pre- and postsynaptic spikes depends on their precise time difference. This figure is taken from [SG10] (with small modifications).

Singularity at $\Delta t = 0$ The rule above is silent about the case of a zero pre-post delay. In biological systems this case can not occur because spikes are not DIRAC pulses but have a non-zero width and thus overlap instead of creating this singularity[5]. However in our model for every postsynaptic spike there is at least one presynaptic spike which occurred exactly at the same time. As this spike is the ultimate cause for the precise time of the postsynaptic spike and we want the STDP to reflect the causal structure, I—like GERSTNER and KISTLER, 2002 among many others—chose to extend the potentiation case to $\Delta t = 0$:

$$W(0) := A_+. \tag{2.7}$$

Spike pairing From the definition above it is unclear which spike pairings are considered for the learning rule. A canonical extension would be to consider all possible pairs, like

$$\Delta w_{ij} = \sum_{t^{\text{pre}} \in T_j^{\text{pre}}} \sum_{t^{\text{post}} \in T_i^{\text{post}}} W(t^{\text{post}} - t^{\text{pre}})$$

with T_i^{post}, T_j^{pre} being the set of all spikes of neuron i and j. But this rule is in violation of experimental results [ID03, MDG08]. A better fit results from using the nearest neighbor pairing [MDG08]: only the ultimate last presynaptic *or* next postsynaptic spike is considered for pairing. Figure 2.3 illustrates the three possible nearest neighbor rules allowed by the 'or'.

It should be noted that STDP rules beyond pairs of spikes exist and are experimentally justified [SG10, MDG08]. Namely the triplet model (interaction between

[5] Beyond that the metaphysical axiom of continuity would forbid singularities in the fine structure of the STDP curve.

two postsynaptic and one presynaptic spike) [SG10, MDG08] and the suppression model (introduction of a STDP-modulating spike efficacy variable depending on the precise time of presynaptic spikes) [MDG08].

However, in my model I use the presynaptic centered nearest neighbor pairing depicted in Figure 2.3b. The weight change then is defined by

$$\Delta w_{ij} = \sum_{(t^{\mathrm{pre}},t^{\mathrm{post}})\in P_{ij}} W(t^{\mathrm{post}} - t^{\mathrm{pre}}) \tag{2.8}$$

$$P_{ij} := \{(t,t') \in T_{ij} \times T_i : \nexists\, \tilde{t} \in T_i : t < \tilde{t} < t'\} \tag{2.9}$$

where P_{ij} is the set of all eligible spike pairs.

To underline the biological plausibility of this rule it can be reformulated into an online version depending only on a finite set of variables and the current event (spike arrival or emission) [SG10]. To this end, two additional variables are introduced: a presynaptic $x_{ij}(t)$ and postsynaptic $y_i(t)$ trace. The weight change can then be formulated with the following equation system:

$$\Delta w_{ij}(t) = \begin{cases} A_+ x_{ij}(t) - A_- y_i(t) & \text{if } t \in T_j \wedge t \in T_{ij} \\ A_+ x_{ij}(t) & \text{else if } t \in T_{ij} \\ -A_- y_i(t) & \text{else if } t \in T_j \\ 0 & \text{otherwise} \end{cases} \tag{2.10}$$

$$\tau_+ \frac{\mathrm{d}}{\mathrm{d}\,t} x_{ij} = -x_{ij} \tag{2.11}$$

$$\Delta x_{ij}(t) = \begin{cases} -x_{ij} & \text{if } t \in T_j \\ +a_{ij} & \text{if } t \in T_{ij} \end{cases} \tag{2.12}$$

$$\tau_- \frac{\mathrm{d}}{\mathrm{d}\,t} x_i = -y_i \tag{2.13}$$

$$\Delta y_i = -y_i + a_- \tag{2.14}$$

The presynaptic spikes ultimately causing postsynaptic spikes ($T_i \cap T_{ij}$) are not covered by equation 2.12 as intended. We want their trace x_{ij} to be 0 after a postsynaptic spike, but use $x_{ij} + a_+$ for evaluation of Δw_{ij}. To implement this, they are updated twice during simulation. Note that such finite online formulations exist for all-to-all and all three nearest neighbor pairing rules [MDG08].

The moment of application of the weight change is critical to maintain causality: a weight update must not depend on a future spike. Care is also required to prevent a circular dependency between a postsynaptic spike and the weight change it may induce. In accordance to Gerstner and Kistler, 2002 potentiation occurs instantaneously at the creation of the postsynaptic spike and depression occurs instantaneously when the next presynaptic spike arrives after that. In both cases the weight update happens after the weight dependent update of the membrane potential.

Limits of weight change are necessary because of two reasons. First the quality of a synapse of being either excitatory or inhibitory does not change in the brain. For our model I further require that connections between neurons neither emerge nor

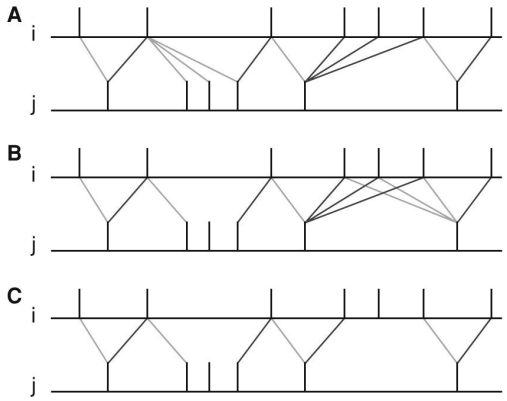


Figure 2.3: Different nearest neighbor spike pairing schemes: *(A)* symmetric interpretation, *(B)* presynaptic centered interpretation, and *(C)* reduced symmetric interpretation. Graphic copied from Morrison, Diesmann, and Gerstner, 2008.

dissolve. Thus

$$sign(w_{ij}) = \text{const} \tag{2.15}$$

is required for all weight changes. The metaphysical constraint of finiteness demands that weights have an upper bound:

$$|w_{ij}| < w_{max}. \tag{2.16}$$

Beyond that I arbitrarily (but in accordance to Izhikevich, 2007 and preliminary experiments) choose to disable any plasticity for synapses from and to inhibitory neurons.

2.1.3 Dopamine-modulated STDP

The STDP rule given above, allows at most unsupervised learning as no reward or error signal exists. As indicated by biological experiments the neurotransmitter dopamine may fit in this role [MHC04, MMB+04, SLF+09]. To transcend unsupervised learning, dopamine has to have an effect on the network's plasticity rule. Thus in accordance with Legenstein, Pecevski, and Maass, 2008 ; and Izhikevich, 2007 (among others) we modulate the STDP intensity with the dopamine signal. How to achieve this is detailed in the rest of this section.

The basic idea is to shift the weight change induced by STDP on the weight w_{ij} eligibility trace c_{ij}, which stores what the STDP would have done to the weights [LPM08]. By slowly decaying, c_{ij} is dominated by recent STDP events. The actual weight change is then induced by the product of

$$\frac{\mathrm{d}}{\mathrm{d}\,t}\, w_{ij} = c_{ij}(d - d_0). \tag{2.17}$$

The rest of this section describes the dynamics of eligibility trace and dopamine level.

The Dopamine signal $d(t)$ is a global variable equal at all synapses in our model. It exponentially decays towards a baseline level[6] d_0 [Izh07]:

$$\frac{\mathrm{d}}{\mathrm{d}\,t}\,d = (d_0 - d)/\tau_d \tag{2.18}$$

$$d, d_0 \geq 0. \tag{2.19}$$

In this situation positive rewards lead to $d > d_0$ which in turn amplifies the STDP. Negative rewards lead to $d < d_0$, reversing the effect of the STDP on the weights. In the trainer no negative rewards are used. This allows us to set $d_0 = 0$ and simplify equation 2.17 and the decay term (2.18) to:

$$\frac{\mathrm{d}}{\mathrm{d}\,t}\,w_{ij} = c_{ij}d \tag{2.20}$$

$$\frac{\mathrm{d}}{\mathrm{d}\,t}\,d = -d/\tau_d. \tag{2.21}$$

Reward is delivered to the system by an instantaneous change of the dopamine level

$$\Delta d(t_{reward}) = R. \tag{2.22}$$

R needs not to be a constant, but can be varied by the learning task. Its details are described in section 2.2. It should be noted here, that to match biological experiments the dopamine change R is more likely to be identified with the reward prediction error—the difference between expected reward and actual reward—than with the reward itself [Sch07a]. Vaguely speaking this causes LTP (LTD) only to occur, if the previous neural activity performed better (worse) than the average activity on the given task. Note also that in this case the dopamine signal complies with the principal characteristics of teaching signals of efficient reinforcement learning models [Sch07b] and with some adaption would allow implementing reinforcement learning within our neural network model [US09].

The eligibility trace c_{ij} is a variable of each synapse with the purpose of caching the result of STDP until it is required for learning by a nonzero dopamine signal. It allows temporal decoupling an action (the spike pair) from its evaluation (dopamine level jump) without loosing the ability to learn in dependency of both [Izh07]. In the mathematical framework of [LPM08] c_{ij} computes as the sum of weighted, temporally shifted eligibility functions $f_c(t) : \mathbb{R} \to \mathbb{R}$ over all relevant spike pairs as

$$c_{ij}(t) = \sum_{(t^{\mathrm{pre}}, t^{\mathrm{post}}) \in P_{ij}} f_c(t - \max(t^{\mathrm{post}}, t^{\mathrm{pre}}))\mathrm{STDP}(t^{\mathrm{post}} - t^{\mathrm{pre}}) \tag{2.23}$$

$$t < 0 \Rightarrow f_c(t) = 0, \tag{2.24}$$

[6] Biologically this could be implemented by a tonic dopamine supplemented with a phasic dopamine response upon reward situations [Izh07]. A more complex biophysical model involving Michealis-Menten kinetics is proposed by [MMB+04].

where $STDP(\Delta t)$ scales f_c by the rules of equation 2.10 of the previous section. They chose a whale back shaped function for f_c. I however use the simpler

$$f_c(t) = \begin{cases} exp(-t/\tau_e) & \text{if } t >= 0 \\ 0 & \text{if } t < 0 \end{cases} . \tag{2.25}$$

We can then achieve an online rule for c_{ij}, because of the pseudoadditivity of f_c under time shift:

$$\begin{aligned} \alpha f_c(t) + \beta f_c(t + \Delta t) &= \alpha e^{t/\tau_e} + \beta e^{(t+\Delta t)/\tau_e} \quad | \, \beta' = \beta e^{\tau_e/\Delta t} \\ &= \alpha e^{t/\tau_e} + \beta' e^{t/\tau_e} \\ &= (\alpha + \beta') f_c(t), \end{aligned} \tag{2.26}$$

which allows us to reformulate the eligibility trace equation 2.23 using an ODE with a well defined jump discontinuity for every spike pair:

$$\frac{\mathrm{d}}{\mathrm{d}\, t} c_{ij} = -c_{ij}/\tau_e \tag{2.27}$$

$$\Delta c_{ij}(t) = \begin{cases} A_+ x_{ij}(t) - A_- y_i(t) & \text{if } t \in T_i \wedge t \in T_{ij} \\ A_+ x_{ij}(t) & \text{else if } t \in T_{ij} \\ -A_- y_i(t) & \text{else if } t \in T_j \\ 0 & \text{otherwise} \end{cases} . \tag{2.28}$$

This equation system is equal to 2.23-2.25 because of the following inductive reasoning: between two adjacent jump discontinuities the integration of 2.27 yields $\alpha e^{t/\tau_e}$ (for some $\alpha \in \mathbb{R}$) and is thus compatible with f_c. Given we could describe a sum of n weighted eligibility functions f_c whose peak was before $t < \tilde{t}$ by setting α correctly, we could also describe $n + 1$ functions, with the peak of the last function at $\tilde{t}$ by applying equality 2.26. Position and height of each jump discontinuity is equal in 2.23 and 2.27, because at most $|\{t\} \cap T_j| + |\{t\} \cap T_{ij}| \leq 2$ summands of 2.23 have their singularity at a given time point and 2.27 covers all resulting 4 cases. This makes the induction step. The base case of $n = 0$ is given with $\alpha = 0$ in accordance to an initial value of $c_{ij}(0) = 0$. Note, that this inductive reasoning is possible at all, because for $t' < 0$ (and thus before their singularity, on which we induce) all $f_c(t')$ are zero and can be dropped.

Integral form As the integral form of w_{ij} is not totally obvious it will be derived here. Given $\mathcal{R}$ is the set of all reward time points (discontinuities of d), then the set of discontinuities of w_{ij} is $T_j \cup T_{ij} \cup \mathcal{R}$. Between two adjacent discontinuities at t_0 and t_1 the integral of equation 2.20 is:

$$\begin{aligned} w_{ij}(\tilde{t})\big|_{\tilde{t}<t_1} &= w_{ij}(t_0) + \int_{t_0}^{\tilde{t}} (\frac{\mathrm{d}}{\mathrm{d}\, t} w_{ij}) \, \mathrm{d}\, t \\ &= w_{ij}(t_0) + \int_{t_0}^{\tilde{t}} c_{ij} d \, \mathrm{d}\, t \end{aligned} \tag{2.29}$$

$$
\begin{aligned}
&= w_{ij}(t_0) + \int_{t_0}^{\tilde{t}} c_{ij}(t_0) e^{-(t-t_0)/\tau_e} d(t_0) e^{-(t-t_0)/\tau_d} \,\mathrm{d}\,t \\
&= w_{ij}(t_0) + c_{ij}(t_0) d(t_0) \int_{t_0}^{\tilde{t}} e^{-\frac{\tau_d+\tau_e}{\tau_d\tau_e}(t-t_0)} \,\mathrm{d}\,t \\
&= w_{ij}(t_0) - c_{ij}(t_0) d(t_0) \frac{\tau_d\tau_e}{\tau_d+\tau_e} \left[e^{-\frac{\tau_d+\tau_e}{\tau_d\tau_e}(t-t_0)} \right]_{t=t_0}^{\tilde{t}} \\
&= w_{ij}(t_0) + c_{ij}(t_0) d(t_0) \frac{\tau_d\tau_e}{\tau_d+\tau_e} (1 - e^{-\frac{\tau_d+\tau_e}{\tau_d\tau_e}(\tilde{t}-t_0)}).
\end{aligned}
$$

2.1.4 Weight normalization

Despite enforcing weight bounds (eq. 2.16) on individual synapses, our plasticity rule can still degenerate the network: unless the eligibility trace is precisely tuned all weights may become extremal. A common way to mitigate these effects is weight normalization: the total weight projecting into a neuron is held constant:

$$\sum_j w_{ij} = W_j = \text{const.} \tag{2.30}$$

Beyond a stability mechanism, weight normalization is a computational tool in itself: for time discrete neural networks it has been shown to yield principal and independent component analysis (PAC, INCA) when used with HEBBIAN learning [Oja82, KOW+97]. It the time discrete case it can be implemented by OJA's rule [Oja82]

$$w_{ij}(t+1) = \frac{w_{ij}(t) + \Delta w_{ij}(t+1)}{\left(\sum_k (w_{kj}(t) + \Delta w_{kj}(t+1))^p \right)^{1/p}} W_j^{1/p} \tag{2.31}$$

with any $p \geq 1$ characterizing the used p-norm[7]. The canonical adaptation to our continuous-time model is to apply this rescaling each time a spike arrives at a synapse and thus causes a weight change—although the weight changes continuously in between, we only need to know it when updating the membrane potential.

But this would force us to perform a computationally expensive summation over all synapses for each incoming spike. Avoiding this requires us to allow temporary violations of equation 2.30. So the instantaneous weight sum S_i is stored in the neuron and together with the known initial weight sum $S_i^0 = W_i$ this allows us to rescale only the synapse affected by an incoming spike:

$$\Delta w_{ij}^{\text{norm}} = w_{ij}^{\text{norm}} \frac{S_j^0}{S_j} - w_{ij}^{\text{norm}} + \Delta w_{ij} \tag{2.32}$$

$$= w_{ij}^{\text{norm}} \left(\frac{S_j^0}{S_j} - 1 \right) + \Delta w_{ij} \tag{2.33}$$

[7] OJA, 1982 used $p = 2$ to implement a PCA, whereas KARHUNEN *et al.* required $p = 3$ to achieve an ICA. The weight normalization alone works for any $p \geq 1$.

$$\Delta S_j = w_{ij}^{\text{norm}}(\frac{S_j^0}{S_j} - 1) + \Delta w_{ij} \tag{2.34}$$

$$S_j^0 = S_j(0) = \sum_i w_{ij}^{\text{norm}}(0) \tag{2.35}$$

$$\tag{2.36}$$

with Δw_{ij} referring to the intended weight changed induced by the previously described plasticity rules and w_{ij}^{norm} the normalized weight.

By comparing equations 2.33 and 2.34 we see that $S_j = \sum_i w_{ij}^{\text{norm}}$ holds during the entire simulation, if the initial weight sum is set up according to eq. 2.35. For $\Delta w_{ij} \approx 0$ (a condition met during most of the time, as dopamine reward is scarce) and with the knowledge that we apply weight normalization only to those positively weighted synapses that are affected by synaptic plasticity mechanisms, eq. 2.34 implies that S_j asymptotically converges to S_j^0. Note that this weight normalization scheme easily allows us to incorporate our weight bounds (eq. 2.16). See section 8.3 for the implementation.

Besides this theoretical argument, the preservation of the weight sum has also been empirically verified during simulations of the model.

2.1.5 Intrinsic Plasticity

The recurrent network's dynamic is highly dependent on the stability of its own activity. In preliminary experiments a very fine tuning was necessary to keep the network in the desire frequency range. To reduce this parameter sensitivity (and make learning possible at all, as we will see in section 3.4) I augment the model with intrinsic plasticity.

Intrinsic plasticity describes processes that change the neurons excitability—in contrast to the aforementioned plasticity mechanisms, which all modify the synaptic transmission efficacy [Tri05]. This form of plasticity has been observed in biological neurons [VWVHW04]. Beyond its role as homeostatic mechanism, it has been motivated by showing that it maximizes information transmission under a fixed energy budget [Tri07]: the entropy of distributions with fixed mean (a fixed energy budget) is maximized for an exponential distribution. The Triesch IP rule thus works by matching the first statistical moments of the observed distribution to that of an exponential [Tri05, Tri07].

We achieved this for our model by changing the firing threshold from u_{thresh} to a neuron-specific value $u_{\text{thresh}} + \Lambda_{1,i}$ and dividing the applied weight for each incoming spike by $\Lambda_{2,i}$ ($\Delta u_i = w_{ji}/\Lambda_{2,i}$). After each evoked spike the IP coefficients $\Lambda_{1,i}$ and $\Lambda_{2,i}$ are updated by

$$\Delta\Lambda_{k,i} = \lambda_k \int_{t_0}^{t_1} f_i^k - \mu_k \,\mathrm{d}\, t \tag{2.37}$$

$$= \lambda_k (t_1 - t_0)((t_1 - t_0)^{-k} - \mu_k) \tag{2.38}$$

for $k = 1, 2$; the instantaneous firing rate $f_i = (t_1 - t_0)^{-1}$; and the k-th statistical moment μ_k of the desired exponential distribution.

Using simulations this rule has been empirically verified to produce the desired

Table 2.1: Valid ranges for the synaptic weight and transmission delays depending on the type of source and target neuron

		postsynaptic			
		excitatory		inhibitory	
		$w_{ij}\,[mV]$	$\tau_{ij}\,[ms]$	$w_{ij}\,[mV]$	$\tau_{ij}\,[ms]$
presynaptic	excitatory	$[0, 4]$	$[1, 5]$	$[0, 3.6]$	$[0.1, 0.5]$
	inhibitory	$[0, 3]$	$[0.1, 0.5]$	$\{0\}$	

output characteristics ($\langle f_i^k \rangle \approx \mu_k$ for $k = 1, 2$) for individual neurons after a settling period. But applied to a whole network of such neurons it produced strong oscillations and put the model into a pathological regime (see section 3.4). In consequence I choose to disable matching the second moment ($\lambda_2 = 0$), ultimately reducing the intrinsic plasticity to a mechanism adjusting a neuron's firing threshold to achieve a desired mean frequency.

2.1.6 Network topology

So far we have only covered aspects of the building block of neural network: the neuron. But the specific choice of connection topology—encoded in synaptic weights w_{ij} and delays τ_{ij}—is at least equally important for the network dynamics. Topology determines if the overall behavior will be dead, chaotic or Turing complete [CR07]. Beyond existence at all also the precise weight distributions matters at lot: Brunel was required to finely tune the weights of his working memory model using mean field analysis [BW01]. Izhikevich argues that the same attention should be drawn to the precise distribution of the transmission delays, because this enables delay equations with infinite dimension and shows qualitatively different behavior in a model with biologically inspired synaptic delay distribution [Izh06]. For our model I consider the topology as well as precise delay *and* weight distribution as essential properties and aim to model them biologically plausible resembling a cell ensemble of the neocortex.

Our model consists of 1000 neurons—an order above the neuron count of a cortical minicolumn and below a cortical column [Rak08]. 800 are excitatory and 200 inhibitory, resembling the cortical proportion of pyramidal neurons and interneurons [BS98]. Two neurons are connected ($w_{ij} \neq 0$) with a 0.1 probability, yielding 100000 synapses or about 100 incoming and outgoing synapses per neuron. Note that this leads to a highly recurrent network: for each neuron pair there is a 1% chance of forming a direct feedback loop.

If a connection ij exists, its weight and delay is equally distributed in an interval that depends on the type of source and destination. The precise values can be found in table 2.1.

The excitatory delay values mimic cortico-cortical axonal delays [Izh06]. The tenfold reduction of inhibitory delays is motivated biologically [BS98] and by experience from our model. Without the delay asymmetry the network would experience repeated patterns of a runaway reaction producing a dead network caused by a collective refractory period occurring after all neurons fired within a $10ms$ window.

Table 2.2: Model constants

description	symbol	value
resting potential		0
firing threshold	u_{thresh}	$15\,mV$
membrane potential decay	τ_u	$50\,ms$
absolute refractory period	$\tau_{\text{refractory}}$	$1\,ms$
transmission delay	τ_{ij}	*per synapse*
dopamine decay	τ_d	$5\,ms$
learning window (LTP)	τ_+	$14\,ms$
learning window (LTD)	τ_-	$34\,ms$
eligibility trace decay	τ_e	$100\,ms$
LTD coefficient	A_+	$1.03 \cdot 10^{-4}$
LTP coefficient	A_-	$0.55 \cdot 10^{-4}$
LTD coefficient	a_+	1
LTP coefficient	a_-	1
maximal weight	w_{max}	$4\,mV$
target frequency		$10\,Hz$
IP coefficient (mean)	λ_1	$5 \cdot 10^{-5}$
IP coefficient (variance)	λ_2	0

The synaptic weights can not be directly adopted from a biological prototype, because they coarsely abstract different phenomena[8]. Dimension analysis constrains the weight to the unit of the electrical potential. With this motivation, I adopt the range of our weight values to the strength of postsynaptic potentials of biological synapses: on the order of $1mV$ [LSW81]. The exact cap at $4mV$ is motivated by [Izh07].

The precise weight intervals are determined to set the average spiking frequency to $10\,Hz$. This value is within the range of frequencies observed in the cortex [Fus73, RM02]. The target frequency for the intrinsic plasticity is set to $10\,Hz$, too.

2.1.7 Model overview

The model specified step by step during the last sections is summarized in the tables 2.2 and 2.3 below. For readability the weight scaling mechanism and weight bounds have been omitted. They can be found in the translation of the model into the domain specific model description language of the simulator in Appendix 8.3.

[8] Among others: synapse types, ion channel count and properties, neurotransmitter reflux, and electrical properties of the postsynaptic membrane up to the axon hillock.

Table 2.3: Model variables

description	symbol	differential function ($\frac{\mathrm{d}}{\mathrm{d}\,t}x$)	difference function (Δx)	discontinuities	site
dopamine level	d	$-d/\tau_d$	external	$\mathcal{R}$ (external)	global
membrane potential	u_i	$-u_i/\tau_u$	$\begin{cases} -u_i & \text{if } t \in T_i \\ w_{ij}/\Lambda_2 & \text{otherwise} \end{cases}$	$\bigcup_j T_{ji} \cup T_i$	neuron
IP coefficient ($k = 1, 2$)	Λ_k	0	$\lambda_k \int_{t_0}^{t_1} f_i^k - \mu_k \,\mathrm{d}\,t$	T_i	neuron
LTD trace	y_i	$-y_i/\tau_-$	$-y_i + a_-$	T_i	neuron
LTP trace	x_{ij}	$-x_{ij}/\tau_+$	$\begin{cases} -x_{ij} & \text{if } t \in T_j \\ +a_{ij} & \text{if } t \in T_{ij} \end{cases}$	$T_{ij} \cup T_j$	synapse
eligibility trace	c_{ij}	$-c_{ij}/\tau_e$	$\begin{cases} A_+ x_{ij}(t) - A_- y_i(t) & \text{if } t \in T_i \wedge t \in T_{ij} \\ A_+ x_{ij}(t) & \text{else if } t \in T_{ij} \\ -A_- y_i(t) & \text{else if } t \in T_i \end{cases}$	$T_{ij} \cup T_i$	synapse
synaptic weight	w_{ij}	$c_{ij} d$	0	$T_{ij} \cup T_i \cup \mathcal{R}$	synapse

2.2 Learning task

The learning task is modeled after neurophysiological delayed response experiments [Fus73]: One of two cue stimuli is given for a short period (cue period, t_c) after which the subject has to wait (delay period, t_d) until another stimulus signals that it should act on the remembered stimulus. Within a short time span (response period, t_r) after the second stimulus the subject has to perform the correct of two actions—the one associated with the initial stimulus[9]. If the response is correct and in time the subject is immediately rewarded. After a short break (intertrial period, t_i) the experiment may be repeated. Learning among which actions to choose (e.g. learning to push one of two levers instead of drumming against the cage) is typically performed before the experiment takes place [JN37].

The sensory and motor skills to complete such a task are outside the scope of this work. For our model sets of 100 excitatory neurons are randomly selected to receive and emit each input and output signal, respectively. As this corresponds to our (physiologically plausible) neural fan in and out, they can be thought as model-external neurons which project into the simulated network (stimulus) or out of it (response).

To give a stimuli an additional $40\,Hz$ Poisson noise (independent and exponentially distributed interspike timing) with $8\,mV$ amplitude for $t_c = 50\,ms$ is delivered to each neuron of the corresponding input set. To read out the response of the network a readout stimuli is applied in the same way for $t_r = t_c = 50\,ms$ to a different set of 100 excitatory neurons. Afterwards we sum the number of spikes evoked during that time for each output set. The model's response is correct iff. the spike count for the output set corresponding to the last input set has more spikes than all other output sets. A draw is interpreted as wrong answer.

Let I_i, R, O_i be the sets of 100 randomly chosen excitatory neurons for cue stimulus, readout stimulus and response output and rand(t) a random time a equally distributed over $[0, t]$. Then each training trial is executed as follows:

1. Select symbol i to be remembered during the trial.
2. Deliver random noise to all neurons of I_i for t_c seconds.
3. Wait $t_d + \text{rand}(50\,ms)$ seconds
4. Deliver random noise to all neurons of R for t_r seconds.
5. Count spikes for all output sets: $A_k(t) := \sum_{n \in O_k} |T_n \cap [t - t_r, t]|$.
6. If $A_i(t) > A_j(t)$ for all $j \neq i$, increase the dopamine level by δ_d.
7. Wait $t_i + \text{rand}(500\,ms)$ seconds

The training is started after letting the network settle for $50\,s$ and then repeated indefinitely. It should be emphasized that the network has no a priori knowledge

[9] Typically the correct action's association with the initial stimuli is not arbitrary, but can be explained by a homeomorphism. For example a subject has to look exactly at that spot which previously lit up, not at some arbitrary chosen place. This notion of a "meaningful mapping" is inapplicable to neural models of this work's scale.

which neurons are responsible for input and output. And that these sets are not even disjunctive: By choosing the neurons for each set independently, each pair of sets has on average 12.5 neurons in common.

2.3 Causality analysis

For analysis of the model, it is of interest to discover the flow of information from input to output neurons. In our model all task-relevant information is transported by spikes—no other mechanism of information transfer between neurons exists[10]. More precisely, we can make the following two observations about causal influence *during a single delayed response trial*, after assuming that intrinsic plasticity is slow compared to the time of single trial ($\Lambda_k \approx const$). For mathematical brevity we assume $\Lambda_1 = 0$ and $\Lambda_2 = 1$ for the rest of this chapter.

1. During a trial, the dopamine level can be assumed as $d = 0$ for all practical purposes. Thus all synaptic weights w_{ij} are constant. Thus the membrane potential u_i and with it a neurons activity depends only on constant values and the time of incoming spikes.

2. Because the postcondition of an outgoing spike is $u_i(t) = 0$, no information about the past beside the precise value of t is stored in the neuron. This allows to completely explain (the time t of) each spike by the time t' of the preceding outgoing spike of the same neuron and the time of all incoming spikes in between:

$$\begin{aligned}
&\left(t' = \min(T_i \cap (t, \infty))\right) && (2.39)\\
\Leftrightarrow &\sum_{\tilde{i}} \sum_{\tilde{t} \in T_{\tilde{i}i} \cap (t,t']} w_{\tilde{i}i} e^{(\tilde{t}-t')/\tau_u} \geq u_{\text{thresh}}\\
&\wedge \forall t'' \in (t,t') : \sum_{\tilde{i}} \sum_{\tilde{t} \in T_{\tilde{i}i} \cap (t,t'']} w_{\tilde{i}i} e^{(\tilde{t}-t'')/\tau_u} < u_{\text{thresh}}\\
&\wedge t \in T_i\\
=&f(t', T_i \cap \{t\}, \bigcup_{\tilde{i}} T_{\tilde{i}i} \cap (t,t']) && (2.40)\\
=&f(t', T_i \cap \{t\}, \bigcup_{\tilde{i}} \{\tilde{t} + \tau_{\tilde{i}i} : \tilde{t} \in T_{\tilde{i}} \wedge w_{\tilde{i}i} \neq 0\} \cap (t,t']). && (2.41)
\end{aligned}$$

   In plain English, equation 2.39 states that neuron i will fire at t' for the first time since the last spike iff. the membrane potential u_i exceeds u_{thresh} at t'.

Based on these observations we can define a causal digraph G in which two spikes (t, i), (t', j) are connected, iff. the former is required to explain the later as of equa-

[10] The dopamine reward actually is a mechanism of information transport across neurons that does not rely on spikes. But as the reward is only given after each trial, this transport can not be used to solve the delayed response task and is thus not considered in this analysis.

tion 2.41:

$$G := (V, E), \tag{2.42}$$
$$V := \{(t, i) : t \in T_i\}, \tag{2.43}$$
$$E := \{((t, i), (t', j)) : w_{ij} \neq 0 \wedge t \in T_i \wedge t' = \min(T_j \cap [t + \tau_{ij}, \infty))\}. \tag{2.44}$$

From knowing that causation is transitive[11] and that the edges of G exactly recapitulate the requirements for direct causation of equation 2.41, we can conclude that a spike v may only have causal influence on another spike v', if a directed path from v to v' exists in G. In other words: all spikes for which no directed path to v' exists could be suppressed without changing the occurrence or time of v' at all. It has to be emphasized, that the existence of such a path is a necessary, not a sufficient condition for causal influence. We can view the causal graph as upper bound of the influence of spikes on each other.

Unfortunately this graph is very dense[12]. We assume causation solely based on spike times. Thus all spikes of a synapse with arbitrarily small, nonzero weight would be considered causative, although their suppression would almost never have an effect. To improve our upper bound for causation we will consider the contribution of each incoming spike to the potential at the soma u_i at the moment it exceeds the firing threshold.

To this end we annotate spike pairs (v, v') (edges in G) with an "egal"-value, that describes how many arbitrarily selected spike pairs (v'', v') we can suppress, before suppressing the arrival of spike v (deleting edge (v, v')) would influence the occurrence of spike v'.

Definition (q-egal) Given a causal graph $G = (V, E)$ and an edge $((\tilde{i}, \tilde{t}), (i, t)) = (\tilde{v}, v) \in E$, then $(\tilde{v}, v)$ is q-egal iff.

$$\forall q' \leq q \forall S' \subset S : (|S'| + q' \geq |S| \Rightarrow (A(S') = A(S' \cup \{(\tilde{v}, v)\})) \tag{2.45}$$
$$S := \{v' : (v', v) \in E \ \wedge v' \neq \tilde{v}\},$$
$$A(S') :\Leftrightarrow \exists\, t_s \in \mathbb{R} : \sum_{\substack{(t',j)\in S' \\ t'+\tau_{ji}\leq t_s}} w_{ji} e^{(t'+\tau_{ji}-t_s)/\tau_u} \geq u_{\text{thresh}}$$

where S is the set of spikes arriving at i before v but after its predecessor. $A(S')$ is true iff. the set of spikes S would have caused a spike at any time.

By denoting the q-egal-value of an edge using $\mathfrak{q}$ we can introduce a q-reduced graph of $G = (V, E)$, in which all edges that are more than q-egal are deleted:

$$\mathfrak{q}((v, v')) = \min\{q : \text{ is } q\text{-egal in } G\} \tag{2.46}$$

[11] Beside being obvious, this can easily be seen by applying equation 2.41 recursively to itself

[12] The probability that two neurons have a direct connection is 0.1. And with an average of 100 synapses per neuron the chance that two neurons are connected through a third one is $1 - (1 - 0.1^2)^{100} \approx 0.63$. Thus for two spikes at t_1, t_2 the probability of being causally related according to G rises quickly to one if both are temporally separated by significantly more than the average transmission delay ($|t_2 - t_1| \gg \tau_{ij}$)

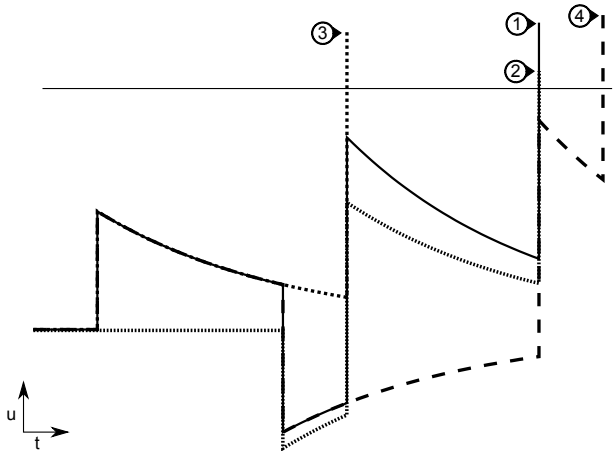


Figure 2.4: A neuron's membrane potential trajectories leading to different time points of spike evocation under suppression of individual incoming spikes. In the undisturbed case (solid line) three excitatory and one inhibitory spike lead to an outgoing spike at (1). Suppressing the first incoming spike (tightly dotted line) has no effect on the outgoing spike (2). Suppressing the inhibitory second spike (dotted line) causes the outgoing spike to occur earlier (3), whereas suppressing the excitatory third spike (dashed line) delays it (4). Except for the first one, all spike arrivals are 0-egal.

$$\mathfrak{q}((v,v')) = \infty :\Leftrightarrow \forall q \in \mathbb{N} : \mathfrak{q}((v,v')) > q \tag{2.47}$$

$$G_q := (V, \{e \in E : \mathfrak{q}(e) \leq q\}). \tag{2.48}$$

The intent of these definitions becomes clearer after some observations and an illustrated example depicted in Figure 2.4:

1. If an edge (v, v') is 0-egal, then deleting it[13] will cause v' not to occur.

2. For each spike, up to q incoming edges e with $\mathfrak{q}(e) \geq q$ can be deleted without altering its occurrence. Any number of ∞-egal edges can be deleted without altering the occurrence of any spike in G.

3. Let $G \prec G'$ denote that G is a sub-graph of G', then

$$G_0 \prec \ldots \prec G_i \prec G_{i+1} \prec \ldots \prec G_\infty \prec G. \tag{2.49}$$

Which graph we use for further analysis depends on our stance on the definition of causality. If we call a spike pair non-causative only if the simultaneous removal of all non-causative spike pairs shall have no effect on the network, then we have to restrict ourselves to G_∞. This view corresponds to a physical interpretation of causality, where nothing outside the past light cone can have an influence on an event, no matter how much the outside universe is perturbed. In other words,

[13]Deletion of edge $((t,i),(t',j))$ is equivalent to suppression of the arrival of a spike (t,i) at synapse ij.

after removing all edges ∞-egal edges from G, the remaining spike pairs would still produce exactly the same trajectory as the original model.

If on the other hand we think the q-egal-value as a quantitative measure related to the resilience against small network perturbations, then G_q contains those spike pairs whose suppression *might* change the network evolution, after (up to) q other arbitrarily choosen spikes from G have been surpressed. In the extreme case G_0 one can not take away a single spike pair without altering the networks behavior. The edges in G_0 can be thought of as the set of most important spike pairs from a q-egal perspective.

Fortunately it is easy to decide which graph to use: Computing G_q for unbounded q is NP-hard. A proof is outside the scope of this work, but the proof idea is to reduce Subset-Sum to computing G_∞. Given a set $S \subset \mathbb{N}$ and a number $T \in \mathbb{N}$, Subset-Sum is the question if

$$\exists\, S' \subset S : S' \neq \emptyset \wedge \sum_{s \in S'} s = T. \tag{2.50}$$

It is known to be NP-hard [CJL+92]. The proof idea is to set $u_{\text{thresh}} = 0$ and to add a spike with weight $-s$ at $t = 0$ for each $s \in S$. At a later time after that a membrane potential of $-T$ would have decayed to almost zero, we add three probe spikes, separated by some delay. The weights of the spikes is arranged so, that the first probe spikes causes a spike if $u(0) \geq T + 1$, the third if $u(0) \leq T - 1$ and the second in the remaining cases $u(0) \in (T - 1, T + 1)$. Due to $u(0)$ being a sum of integers, the second probe spike can only cross the firing threshold if $u(0) = T$. Thus iff. the second spike is not ∞-egal, there exists a subset of S whose sum is T.

In contrast, computing G_0 is possible in $\mathcal{O}(|E| \log |E|)$ with $|E|$ being the number of edges in G—the number of spike arrivals in the simulation. To this end we normalize all spike arrivals to unit weight $w_{ij} = \pm 1$ by shifting their arrival time t to $t + \Delta t$:

$$\begin{aligned} |w_{ij} e^{-\Delta t / \tau_u}| &= 1 \\ \Rightarrow \Delta t &= \tau_u \log |w_{ij}|. \end{aligned} \tag{2.51}$$

By equation 2.1 such a normalized spike arrival has the same effect on the membrane potential as the original one for any $t' \geq t$.

The devised algorithm then works as follows: For each neuron, it iterates over all spike arrivals in temporal order. If a spike arrival is inhibitory, then suppressing it can not cause an immediate spike but potentially allow a later excitation to reach the firing threshold. We thus store the time of the normalized spike in the temporally-sorted binary tree B_{inh}. At each excitatory spike arrival we compute the minimal arrival time of a normalized inhibitory spike whose suppression would cause our neuron to fire now. Every inhibitory spike arrival in B_{inh} with an equal or higher time is marked 0-egal and removed from the tree. We then store the normalized time of the excitatory spike in binary tree B_{exc} and proceed to the next spike. Once we reach the spike arrival that causes the membrane potential to exceed the firing threshold, we mark it 0-egal and compute the minimal time of a normalized excitatory spike whose suppression would cause our neuron to not fire now. Every excitatory spike arrival in B_{exc} with an equal or greater time is marked 0-egal and

removed from the tree. The remaining elements in B_{inh} and B_{exc} are not 0-egal and can be removed from G_0. For each edge in G the algorithm requires two insertions, two deletions and two lookups in binary trees. The binary tree size is bounded by $|E|$. Enumerating all edges of the graph for iteration in temporal order is possible in $\mathcal{O}(1)$ per edge. By maintaining one B_{inh} and one B_{exc} for each neuron, spikes of all neurons can be marked in parallel. The time requirement is thus $\mathcal{O}(|E| \log |E|)$.

3 Results

The neural network model developed in chapter 2 has been numerically simulated. It achieves a small but significantly above-chance performance in the posed delayed-response tasks after a brief learning period. The preconditions, characteristics, and mechanisms of this ability are illuminated in the following sections.

In section 3.1 the detailed dynamics of a single simulation run are examined. The causal digraph of a simulation run is analyzed in section 3.2. Section 3.3 gives an overview of the performance under varied tasks, answering the question for how long information can be retained. Finally, different model simplifications are proposed. Their lack of achievement is summarized in section 3.4.

The software developed for simulation and analysis is described in Appendix 8.4, together with a design rationale and highlighted code excerpts. The simulator offers a domain specific language (DSL) in which all model variants are expressed (see Appendix 8.2). The implementation of the mathematical model description of chapter 2 in this DSL can be found in Appendix 8.3.

3.1 An exemplary run

Figure 3.1 shows the performance development of the default model, averaged over 8 runs with different initial random seeds as well as a single run, which will be analyzed in detail in this section. After $500\,s$ and on average 327 task trials, those 8 simulations correctly answered 53.5% of the delayed response tasks during the next $500\,s$. Beyond the first $1000\,s$ the accuracy remained constant.

This result is small, but significantly above chance. During $500\,s$ of simulation, on average 367 task trials are executed, 2909 in total over 8 simulations. The performance of randomly guessing is described by a BERNOULLI process: n independent trials of equiprobable outcome 0 (wrong guess) or 1 (correct guess). So the number of correct guesses k over n trials is binomially distributed with mean $\frac{1}{2}n$ and standard deviation $\frac{1}{2}\sqrt{n}$. Chance performance is thus of mean $\mu = 0.5$ and standard deviation $\sigma = \frac{1}{2n}\sqrt{n} = 0.65\%$. The achieved performance is 5.4σ above chance performance.

The simulation with random seed 0, depicted in 3.1b achieved a mean performance of 57.4% (4.0σ) during $500\,s \leq t \leq 1000\,s$, making it a perfect candidate for detailed analysis. In Figure 3.2 the time course of each model variable of a representative synapse and neuron is shown for $1000\,s$ simulation time, as well as a $1\,s$-period during which the network receives a stimuli and gives the correct answer. The chosen neuron belongs to the wrong-output set for this stimuli, but no other set. The shown synapse projects from another wrong-output neuron to this one.

For the selected neuron, the intrinsic plasticity coefficient settles after $50\,s$ to $\Lambda_1 \approx -7\,mV$, reducing the effective firing threshold to $u_{\text{thresh}}+\Lambda_1 \approx 8\,mV$. Nonethe-

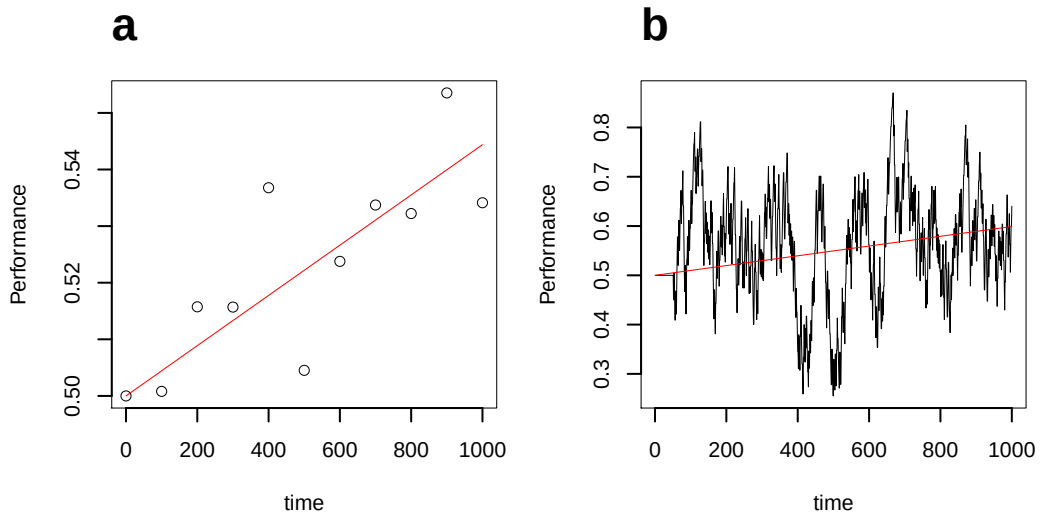


Figure 3.1: Model performance over 1000 s training, including 692 task trials: *(a)* mean performance, average over 100 s-intervals and 8 simulations with different random seed; *(b)* exponential performance average ($\lambda = 0.9$) of a single simulation (seed: 0). Both plots include a linear regression with y-intersection held at 0.5.

less the network is dominated by inhibitory activity, keeping the mean membrane potential at $-8.2\,mV$.

The total weight change of the portrayed synapse during the 1000 s training interval is $+0.30\,mV$ or $+7.5\%$ of the maximal weight. The weight rises continuously over the whole training period, although 24% of the weight change are caused by 10 jumps. In the 1 s-plot, the weight change effects of the weight normalization can be seen: only one of the many jumps occurred due to a dopamine triggered takeover of the eligibility trace to the synapse's weight. The major weight changes are mediated by weight normalization. The weight seems to change even when the dopamine level is zero, but this is an simulation artifact[1].

In the shown 1 s-time frame, the neuron fires neither during input nor readout phase, but during readout the membrane potential is significantly increased. The intensive firing prior to the input phase has no connection to any training trial. It is one example of the highly self-exciting activity induced by the recurrent network topology.

[1]For the sake of simulator efficiency, the dopamine reward is not signaled to a synapse until it receives a spike the first time after the reward. The resulting weight sum change at the neuron is signaled to its other synapses only once they receive a spike. In this scheme a synapses own eligibility trace is always correctly accounted for the weight applied a the neuron, but the weight scaling stays $\langle f_i^{-1} \rangle$ behind (with $f = 10\,Hz$ target frequency). Which—given the small magnitude of weight changes—is negligible.

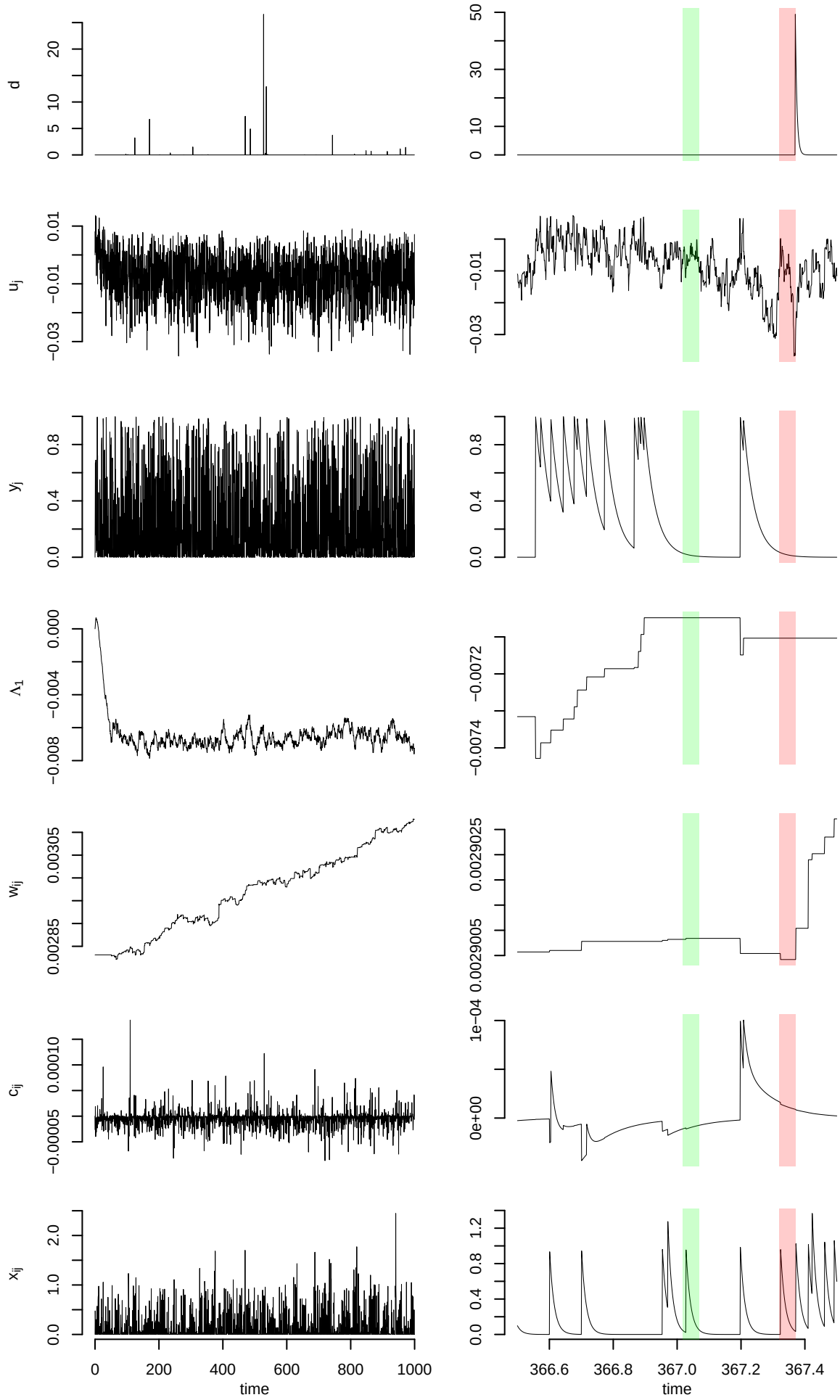

d
u_j
y_j
Λ_1
w_ij
c_ij
x_ij
time
time

Figure 3.2: (Left page) Time course of all model variables for an example instance. The left column shows the whole simulation (time step 0.5 s). The right column shows a 1 s-closeup, during which a trial takes place (time step 0.5 ms). The input and readout period is highlighted green and red, respectively.

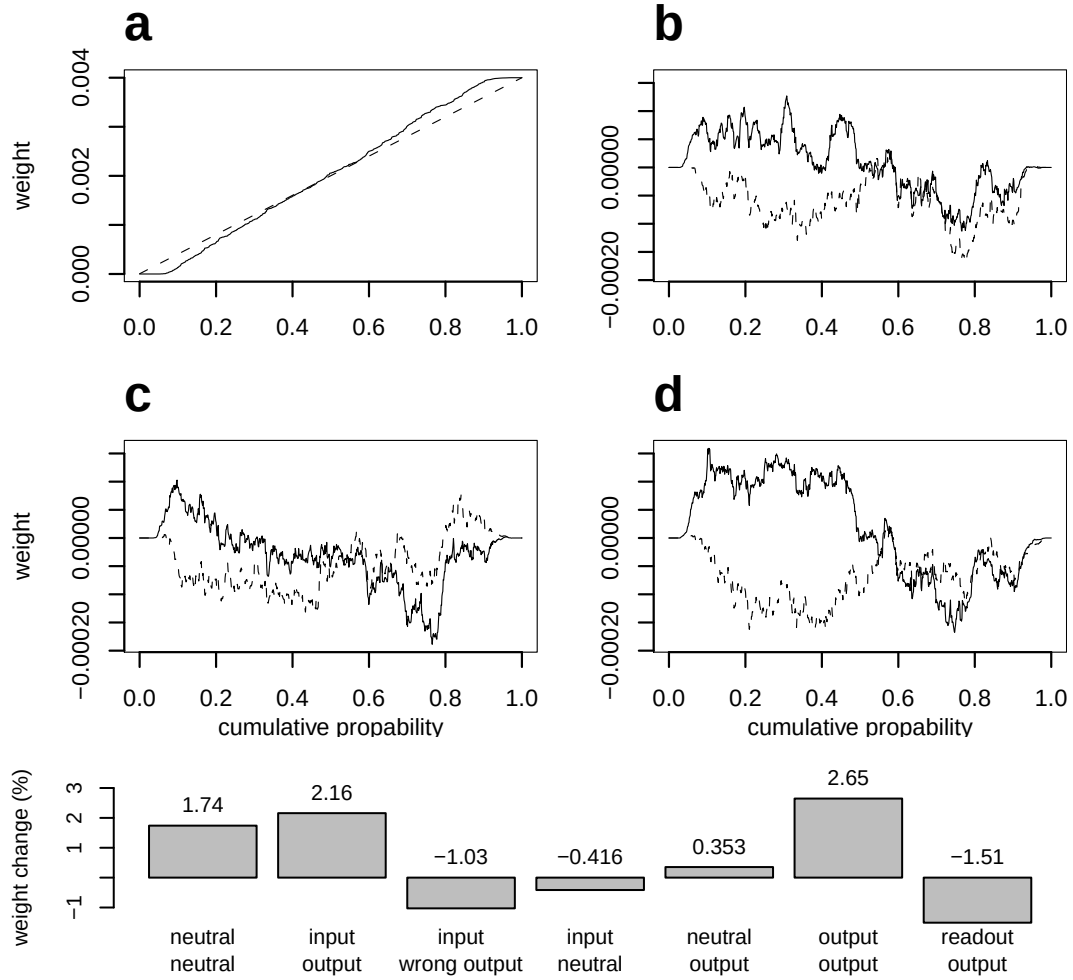


Figure 3.3: Inverted cumulative weight distribution after 1000 s training. *(a)* Weight distribution of all excitatory-excitatory synapses at $t = 1000\,s$ (solid) and $t = 0$ (dashed). In plot *(b)-(d)* the cumulative distribution of *(a)* is subtracted from the shown distributions. They show
(b) input→output (solid), input→wrong output (dashed),
(c) neutral→output (solid), input→neutral (dashed),
(d) output→output (solid), and readout→output (dashed)
synapses, where neutral refers to all excitatory. Each group's weight is sampled over 900 synapses. The relative weight change of each synapse class is shown at the bottom.

To visualize how the repeated training had shaped the network and understand why it achieved an above-chance performance, the cumulative weight distribution

after 1000 s simulation time is illustrated in Figure 3.3 for several groups of synapses. To this end we consider five classes of neurons and the synapses between them: input, readout (receiving the readout stimulus), output, wrong output and neutral. The last refers to all excitatory neurons.

Figure 3.3a shows that the weight distribution becomes more extremal over time: after 1000 s simulation 16.2% of all synapses are within 1% of the extremal values $(0, 4\,mV)$, compared to 2% at $t = 0$. The approximate rotational symmetry of the curve can be ascribed to weight normalization.

In Figure 3.3b one can see that the weight evolution of synapses from input to output neurons is anti-correlated to that of input to wrong-output neurons: input→output connections gain strength, whereas input→wrong-output connections are weakened. As the activity of output neurons has no special influence on the network besides causing the dopamine reward, the later is the only way the network could have learned to differentiate between both groups.

In the three Figures 3.3b-d the relative cumulative weight distribution of all synapse groups is subzero for upper half of the plot, except for input→neutral synapses. This implies that the weight increase for synapses above $2\,mV$ weight occurs primarily for neutral→neutral and input→neutral connections.

The weight gain of output→output-synapses in the 0-2 mV range is strongly above the average (Figure 3.3d). During the simulation it increased by 2.64%the strongest increase of all synapse classes analyzed. The anti-correlated weight sum of readout→output synapses has the strongest weight loss of all: -1.59%.

The weight-increase of intra-group connections for neutral and output neurons hints at the possibility that one of them (or both) may house neuron ensembles that maintain the input signal. For the output neuron group the fact that input→output weights grow, hardens this suspicion: Increased activity is required more than 250 ms after input stimulation. After this time period an elevated membrane potential of output neurons has long been decayed or been reset by emitted spikes.

To examine this hypothesis, we can look at the mean firing rates of three neuron groups during successful trials: neurons of the correct-output group, the wrong-output group and neutral neurons (Figure3.4). During input stimulation all three groups follow the same trajectory rather precisely: fast rising frequency during stimulation and a rapid drop afterwards. The common fate ends 50 ms after the termination of input stimulation. From then on, the frequency of the correct-output neurons is slightly above the still-intertwined trajectories of the other two groups. A second qualitative behavioral change occurs approximately 240 ms after the trial starts: a short frequency jump of all group's firing rate. Afterwards the correct-output neuron's frequency is still elevated above the network's average, but additionally the wrong-output neurons fire less frequently then both other groups. The difference in firing rate between neurons of correct and wrong output is maintained during the readout phase, although both groups show an absolute increase in the firing rate due to readout stimulation.

3.2 Application of causality tracking

To further understand the inner working of the model I have applied the causal graph analysis introduced in section 2.3. To this end the causal graph G_0 has been

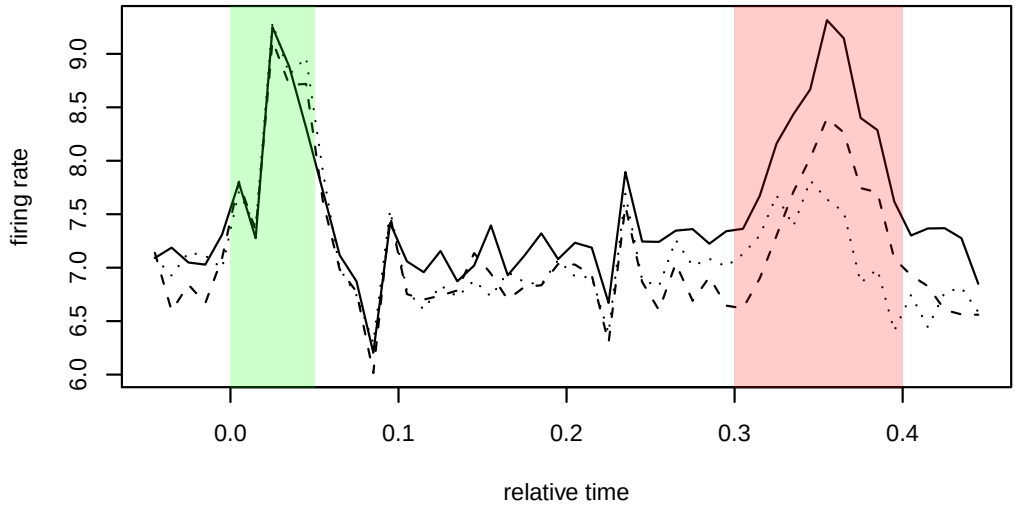


Figure 3.4: Firing rates during correctly solved trials of neurons of the correct output group *(solid)*, the wrong output group *(dashed)* and a 100-neuron sample of all excitatory neurons *(dotted)*. The frequency is computed in 10 ms time bins and averaged over all 352 successful trials. The time is relative to the onset of input stimulation ($t = 0$). Colored background denotes the application of input stimulus *(green)* and the time window during that the 50 ms readout window occurs *(red)*.

computed for the simulation run analyzed in detail in the preceding section.

Let us first examine the fraction Q of 0-egal spike arrivals among the set of all spike arrivals. Initially we find $Q = 29.3\%$ of all spike arrivals to be 0-egal (sampled over $0 \leq t \leq 10\,s$). After the network is settled ($50 \leq t \leq 60\,s$) the fraction is reduced to $Q = 26.8\%$. During training Q slowly decreases. After 940 s training $Q = 23.9\%$ of all spike arrivals are 0-egal ($990 \leq t \leq 1000\,s$). The time course of Q computed for 1 s time bins can be seen in Appendix figure 8.1. One can see that Q is falling fast during the settling period and slowly during the rest of the simulation. The standard deviation of Q between 1 s time bins is 0.0130 or 5.24% of the mean; Computed over 1000 s, the fraction of 0-egal spike arrivals is $Q = 24.9\%$.

In the right part of Figure8.1, we can see the short term dynamic of Q during a trial, computed with 25 ms time bin and averaged over all trials. Due to the high standard deviation, the suggestive increases of Q after input and readout stimulation are highly insignificant.

Finally the causal graph has been colored, to track how information is transmitted from the input neurons during stimulation to the output neurons during the readout phase. To this end, all external spikes belonging to the input stimulus are marked. Then iteratively all spikes are marked that are causally influenced by a marked spike (share an edge with it in G_0). As edges in the causal graph point temporally forward, the procedure is complete once the last spike of the readout phase is reached. One can think of this coloring as the set of causally-relevant spikes, as opposed to the set of causally-relevant spike arrivals we considered so far. The coloring *would* help us to discover the spatiotemporal subset of neuronal activity that holds the input stimulus.

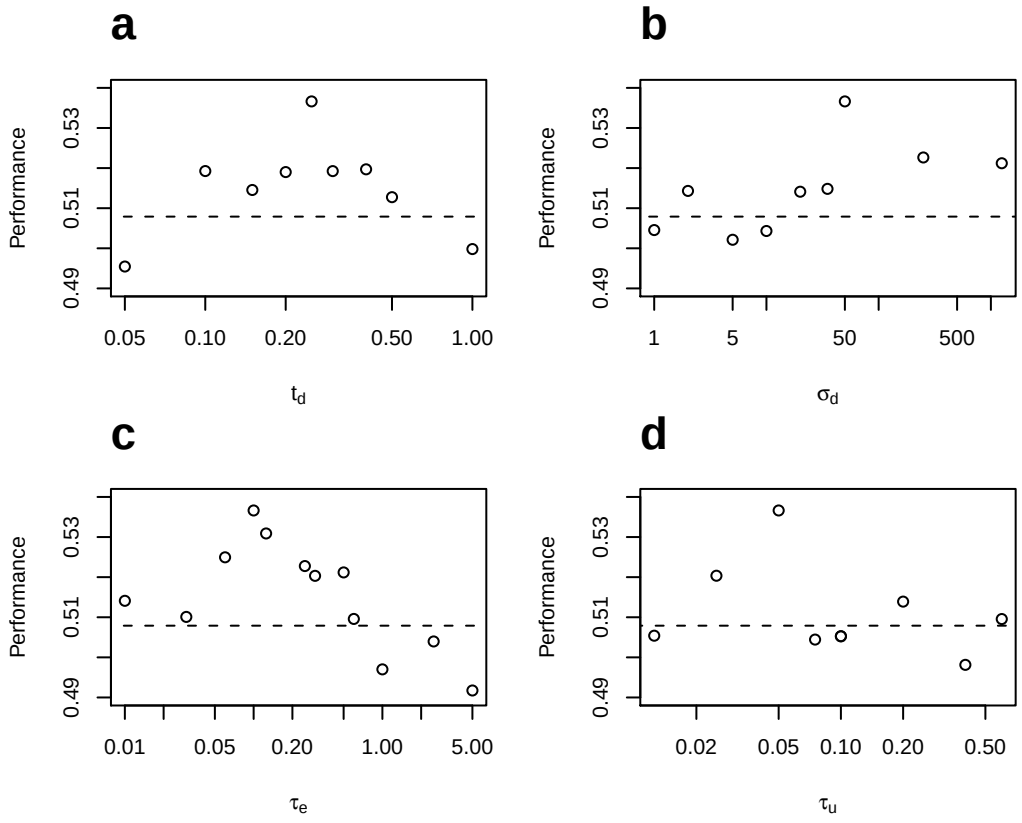


Figure 3.5: Mean performance of 4 runs over 1000 s in dependency of several model constants (left to right, top to bottom): *(a)* delay period t_d, *(b)* dopamine reward δ_d, *(c)* eligibility trace decay constant τ_e, and *(d)* membrane potential decay constant τ_u. The x-axis is logarithmic in all plots. The upper standard deviation of chance performance ($\mu + \sigma$) is marked with a dashed line. The simulated model is the default model except for the varied parameter. In all four cases, the value chosen for the default model exhibits the biggest performance.

Unfortunately, the application of this algorithm to the causal graph of our model leads to marking almost all spikes ($> 99.9\%$ in every trial) as relevant even before readout stimulation is over. Given the nonexistent expressiveness of this coloring a detailed structural analysis of the colored causal graph makes no sense.

3.3 Performance stability

To be of a general purpose, it is crucial that the network's performance is insensitive to small parameter changes and not tuned towards a very narrow class of tasks. To learn more about the ability of the network in this regard, the performance has been measured for various delay periods t_d and dopamine rewards δ_d, and decay constants of eligibility trace τ_e and membrane potential τ_u. The results are depicted in Figure 3.5.

In Figure 3.5a the expected inability store information for long periods ($t_d \geq 1\,s$) can be seen. Surprisingly the network also fails to give the correct response shortly

after receiving input ($t_d = 50\,ms$). Intuitively one would assume delayed-response task difficulty to be monotonically increasing with longer delay periods. Instead we observe a performance peak at $t_d = 250\,ms$.

One explanation for this behavior can be found in Figure 3.5c, which shows the dependency of the performance on the decay time of the eligibility trace. We can again see a performance peak at the default value ($\tau_e = 100\,ms$). Towards smaller values the performance decreases but stays above chance. Towards larger values the performance also decreases but reaches chance performance ($0.6\,s \leq \tau_e \leq 2\,s$) and ultimately falls significantly below chance performance ($\tau_e = 5\,s$). The later extreme case can be easily explained: once τ_e is much larger than the intertrial delay (in this case $(1 \pm 0.25)\,s$), the eligibility trace (c_{ij}) depends almost as much on past trials as on the current one. This allows small differences in the mean LTP and LTD traces ($\langle x_{ij} \rangle$ and $\langle y_i \rangle$) to dominate the weight change. These difference are introduced by random deviations in topology and conductance delays. Ultimately this leads to learning the wrong input-output association in 2 out of 4 runs at $\tau_e = 5\,s$.

The sharp performance decrease on both sides of the peak at $\tau_e = 100\,ms$ leads to the hypothesis that a certain ratio between eligibility trace decay and delay period duration is required to achieve a good performance. To test this hypothesis and answer the question why the network fails trials with very short delay periods a modified model with $\tau_e = 20\,ms$ and $t_d = 50\,ms$ has been simulated for $1000\,s$. The ratio of τ_e to t_d is the same for modified and original model ($\frac{20\,ms}{50\,ms} = \frac{100\,ms}{250\,ms} = 0.4$). It achieved a mean performance of 0.551, supporting the hypothesis.

In Figure 3.5b the amount of dopamine reward σ_d given after a successful trial is varied. Although there is a distinct maximum at $\sigma_d = 50$, the network is able to achieve a performance two standard deviations above chance performance over more then one order of magnitude: $50 \leq \sigma_d \leq 1000$.

In Figure 3.5d the membrane potential decay constant is varied. Besides the performance maximum at $\tau_u = 50\,ms$, it shows two remarkable facts. First, the still significant performance at $\tau_u = 25\,ms$. The entire model has been build around a fixed $\tau_u = 50\,ms$: the balance between inhibitory and excitatory weights is tuned to it and without homeostatic mechanisms the network is extremely sensitive regarding τ_u (see also section 3.4). Albeit halving τ_u, a significant above-chance performance has been achieved. On the other hand, for $\tau_u > 50\,ms$ the performance quickly diminishes to chance-level.

This lack of performance coincides with the networks inability to reach the target frequency of $10\,Hz$. After $1000\,s$ simulation time, the median of the exponential frequency average (sampled with $\tau = 20\,s$) of 100 random, excitatory neurons of the default model ($\tau_u = 50\,ms$) is $9.54\,Hz$. For the varied model with $\tau_u = 200\,ms$ the median is $5.86\,Hz$ and 6 out of 100 sample neurons have not fired once in the entire simulation.

One may assume that this is caused by the one-sided limit of IP coefficient Λ_1 by the following line of thought. The network is dominated by inhibition. Increasing τ_u increases the time it takes until negative charge due to an inhibitory burst fades away. To compensate a lack of activity, Λ_1 has to decrease. But this ability to counteract is limited due to the constraint $\Lambda_1 > u_{\text{thresh}}$. But this hypothesis does not hold: although the Λ_1 distribution is shifted towards the minimum for increased τ_u, the number of neurons close to $\Lambda_1 \approx u_{\text{thresh}}$ does not qualitatively differ (see

Appendix Figure 8.2).

3.4 Necessity of model components

The presented model is quite complex, not only in the high dimension of its parameter space, but already by using three different homeostatic mechanisms: an absolute refractory time (ARF), weight scaling (WS), and intrinsic plasticity (IP). Adhering to Occams razor requires us to test if all these model components are necessary. A proof of the question is unobtainable within the limits of this work. The table below shows all twelve model variants obtained by selectively combining these three homeostatic mechanisms. For intrinsic plasticity two variants are listed: A full IP, in which first and second moment are matched to an exponential distribution. And a mean-only IP, in which only the first moment is matched. The later is the one used throughout this study.

Each model variant has been tested for various parameter combinations and except one each showed the same failure conditions independent of the tested parameters. This can be considered as indication for the simultaneous necessity of all three homeostatic mechanisms.

Table 3.1: Dominant failure conditions observed in different model variants. See below for explanation of abbreviations.

	without weight scaling		with weight scaling	
	no ARF	ARF	no ARF	ARF
no IP	RA	RA, SD	RA	NPI, LPD
mean-only IP	RA, NPI	NPI	RA, NPI	-
full IP	RA, SD	SD	RA, SD	SD

As depicted in the above table, four typical failure conditions have been isolated:

Runaway activity (RA) The instantaneous frequency of the network increases to extremely high values. This happens once the current transported by in-flight spikes surpasses a critical threshold: on arrival they create even more spikes and inhibition either does not adapt fast enough to compensate the increasing excitatory current or has saturated its firing rate due to the ARF. If inhibition is made strong and fast enough to prevent this super-critical excitatory activity, the inevitable increased frequency during input and readout phase cause spontaneous death of the network.

Spontaneous death (SD) The network is primarily driven by recurrent, not external excitation (the external current is much smaller the internal). If this self-supporting stream of internal drive is interrupted—even for a short period—it can result in long periods of almost no network activity (from $100\,ms$ to several seconds). See Figure 3.6 for an example. Although the network eventually recovers due to relaxing IP coefficients and random external spikes, no recurrent activity survives the SD. Thus no information can be transported across

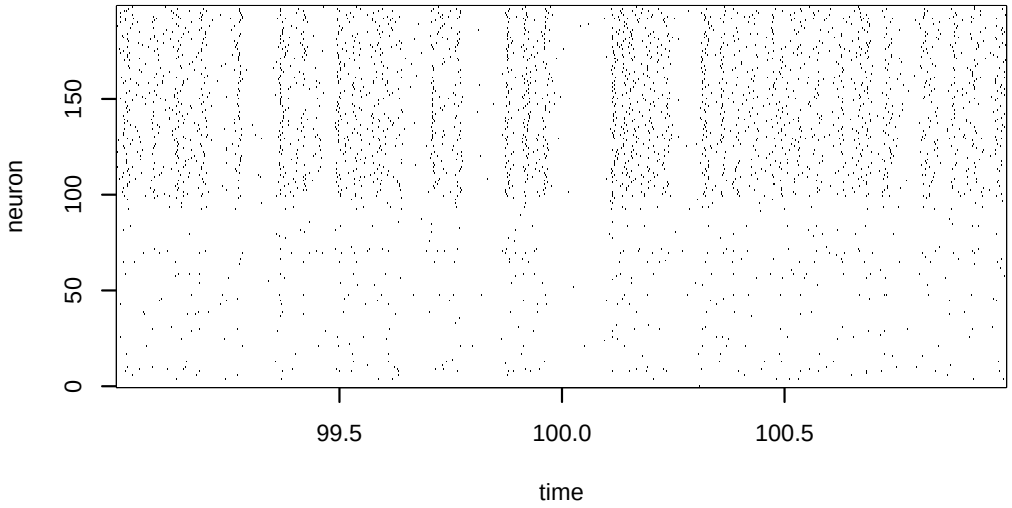


Figure 3.6: Example of repeated spontaneous death in a model variant with absolute refractory time, full IP ($\lambda_1 = 10^{-4}$, $\lambda_2 = 10^{-8}$), no weight scaling and no task-related external stimuli. Shown is the activity of 100 excitatory (0-99) and 100 inhibitory (100-199) neurons for $2\,s$, after the network settled for $100\,s$. During the longest silence period around $t = 100\,s$ only two excitatory neurons fire within a time span of $101\,ms$—both due to coinciding external spikes.

an SD boundary. The repeating occurrence of spontaneous deaths keeps the network in a regime where it is not able to perform its task.

The three observed sources of internal drive interruption leading to spontaneous death have been *(1)* too strong inhibition, *(2)* a preceding runaway activity which was ultimately stopped by intrinsic plasticity rules and *(3)* full IP, driving neurons to a regime where they become very sensitive to deviations of current influx.

No performance increase (NPI) The network performance does not exceed chance level even after extensive training.

Late performance decrease (LPD) The network initially achieves an above-chance performance, but eventually returns to a chance-like performance after several hundred seconds. Note that this failure case has been observed only in a small, carefully tuned parameter region. Outside this region the model variant simply achieved no performance increase.

The only model variant that achieved an enduring, significantly above-chance performance required an absolute refractory time, weight scaling and a mean-only intrinsic plasticity.

4 Discussion

In this work a neural network model for learning delayed response tasks with dopamine-modulated spike-timing dependent plasticity has been proposed. The model has been shown to achieve a small but significant above-chance performance after a brief training period. Due to the small performance increase this result can only be regarded as proof that learning working memory using a dopamine reward is possible, but not as argument that this actually is the case in the brain.

The demonstrated delay period (250 ms) is small compared to biological experiments, which delay the readout several seconds [JN37, Fus73, RM02]. But our model uses the same decay constants for excitatory and inhibitory neurons. Brunel and Wang, 2001 used a different decay characteristic for both synapse times and increased the decay time of excitatory by a factory 10 over the decay time of inhibitory synapses. From a biologically point of view this resembles slow pyramidal neurons and GABAergic fast spiking interneurons, respectively and is thought to be necessary for WM by many authors [BW01, DSS00]. On one hand, this work showed that this is not a strict requirement for solving a delayed-response task with short delay periods. On the other hand, the observed performance might increase if this synapse characteristic is incorporated; From a computational point of view, a slowly decaying influx of excitatory current is a mechanism for information storage between spike emission. Without it, the network entirely relies on in-flight spikes and is thus forced to operate on the time scale of axonal conductance delays.

But this remains speculation, because the mechanism by which the network remembers the input stimuli has not been identified conclusively. This was complicated by the low performance which caused a small signal-to-noise ratio. Analysis of two different measurement categories revealed information about the mechanism: the learned weights after training and the firing rate during a trial.

The performance of the model has been indirectly shown to be very sensitive regarding the weight distribution; The weight change is linear in the dopamine reward and halving the optimal dopamine reward has shown to drastically reduce the response accuracy. This practically rules the model out for explaining how to learn working memory over long time scales—as observed during childhood [GPAW04]. This would require a learning mechanism that is stable against noise in the synaptic weight development. On the other hand this sensitivity underlines the importance of the performed weight analysis, despite the small observed difference over time and between synapse classes.

The weight increase of connections from input, neutral and output neurons to output neurons, together with the slightly elevated frequency of correct-output compared to wrong-output and neutral neurons is an indicator that a self-exciting group of neurons has formed in the group of output neurons. But further investigation is necessary, especially regarding the question why the elevated activity of the output neurons does not start until $50\,ms$ after the input stimulation is terminated and how

the input stimulus is remembered during this gap.

The time course of the firing rate during a trial can not be compared to biological observations directly. As observed in vivo, the firing rate of the network increases during a trial, especially during input and readout phases [RM02]. But this can be attributed to the intense input and readout stimulation and is thus no observation but an effect enforced by the experimenter. Besides that, the observed frequency difference is as low as $2\,Hz$, compared to as much as $35\,Hz$ observed in wet experiments [RM02].

This deficiency of incomparability could be eliminated by changing the training protocol; Instead of applying additional random noise to different groups of neurons for input and readout stimulation, one could maintain a constant-mean random noise and only change the fine structure of the external random spikes. One implementation possibility for this is polychronization—neuron groups exhibiting time-locked spatiotemporal spike patterns [Izh06]. They have been proposed as a mechanism for information transmission and storage in working memory tasks [SI10].

Another change to the trainer worth examining relates to the reward function. In this work a fixed reward was given for each succesfully absolved trial. But research suggest that one role of dopamine is to encode the difference between expected reward and actual reward, framing it as reinforcement signal in a reinforcement learning scheme [MHC04]. Changing the dopamine signal in such a way would also allow to align our model more closely to the theoretical analysis of dopamine-modulated STDP by Legenstein, Pecevski, and Maass, 2008: A zero-mean reward signal is required for the applicability of their results [LPM08].

The most surprising result of this work is that for learning the delayed response task the decay time of the eligibility trace has to be on the scale of the delay period, with the optimal ratio at $\tau_e/t_d = 0.4$. This ratio implies that the trace of STDP events from the beginning of a trial have 8% the size of those immediately before the reward. This can be interpreted as an estimate of the noise resistance of dopamine-reward mediated learning. Izhikevich demonstrated for several tasks that learning with dopamine-modulated STDP is resistant to noise in the eligibility trace that is caused by task-irrelevant spikes, even if the reward comes seconds after the desired activity. Incidentally these results where obtained with a similar decay/delay-ratio of $\tau / \langle t_d \rangle = 0.5$ [Izh07]. Hypothesizing that this optimal ratio is the same in vivo could allow to infer the eligibility trace decay rate from psychometrical experiments. This could be used to narrow the range of substances candidates, that implement the eligibility trace, by comparing their decay times.

The model developed in this work is of a very heterogeneous complexity. This leaves a lot of room for arbitrary decisions which ultimately reduces significance and applicability of the findings. The neuron and synapse model used is the simplest spiking neuron model. Reward mechanism and dopamine dynamics are simple in comparison to the literature, too. Several aspects of the model deviate from the default approaches found in the literature: External random noise is of small frequency and high amplitude, instead the other way around [BW01, Izh07]. The intrinsic plasticity mechanism used is simple but deviates from the known approaches and is not thoroughly examined [SJT10]. The used weight scaling approach is especially delicate; Its complexity widely exceeds the common approach of rescaling all synapses in fixed time intervals. No use of it outside of this work is known to the

author. Although it had been verified to maintain a constant weight sum, its effect on a synapse's weight depends on the precise order and frequency of spike arrivals. Synapses that are more active are more often subject to weight scaling. The effect of this bias has not been examined.

It has been indicated that a simple omission of model components is no viable approach to increase simplicity and with it the expressiveness of the model. This leaves two future directions for development of the model: Either the arbitrariness of the model components is reduced and the model simplified. Or the components are augmented and parametrized with more in vivo data, to obtain a biologically more plausible model.

One to deviate from common approaches for IP and weight scaling has been to maintain the model in a form in which the causal graph analysis could be applied. But this effort has not paid off. Ultimately the method delivered only two insights into model: First, that it is driven in a dynamic regime where the deletion of a single spike arrival leads to a changed neuronal fate with a high probability (25%), although the neuron receives on the order of 1000 spikes per second. And second, that this sensitivity slowly declines while training the network. The later can be explained by the increasing number of synapses with approximately zero weight.

With 25% of all spike-arrivals being 0-egal, the resulting graph is much too dense for further graph-theoretical analysis. Nonetheless, this isolation of the most essential spikes may still be a useful building block for future analytical tools. But to improve the rejection rate of less interesting spikes above the levels achieved in G_0 will be difficult; The q-egal annotation described so far was a spatiotemporally local rule. Future work may use global knowledge to eliminate more redundant spike arrivals.

5 Bibliography

[Bad00] A. Baddeley. The episodic buffer: a new component of working memory? *Trends in cognitive sciences*, 4(11):417–423, 2000.

[Bad03] A. Baddeley. Working memory: Looking back and looking forward. *Nature Reviews Neuroscience*, 4(10):829–839, 2003.

[Bad10] A. Baddeley. Working memory. *Scholarpedia*, 5(2):3015, 2010.

[BGB+98] R.L. Buckner, J. Goodman, M. Burock, M. Rotte, W. Koutstaal, D. Schacter, B. Rosen, and A.M. Dale. Functional-anatomic correlates of object priming in humans revealed by rapid presentation event-related fMRI. *Neuron*, 20(2):285–296, 1998.

[BH74] A.D. Baddeley and G. Hitch. Working memory. *The psychology of learning and motivation*, 8:47–89, 1974.

[BH99] N. Burgess and G.J. Hitch. Memory for serial order: A network model of the phonological loop and its timing. *Psychological Review*, 106(3):551, 1999.

[BRC+07] R. Brette, M. Rudolph, T. Carnevale, M. Hines, D. Beeman, J.M. Bower, M. Diesmann, A. Morrison, P.H. Goodman, F.C. Harris, et al. Simulation of networks of spiking neurons: a review of tools and strategies. *Journal of computational neuroscience*, 23(3):349–398, 2007.

[BS98] V. Braitenberg and A. Schüz. *Cortex: statistics and geometry of neuronal connectivity*. Springer Berlin, 1998.

[BW01] N. Brunel and X.J. Wang. Effects of neuromodulation in a cortical network model of object working memory dominated by recurrent inhibition. *Journal of Computational Neuroscience*, 11(1):63–85, 2001.

[CBGRW00] A. Compte, N. Brunel, P.S. Goldman-Rakic, and X.J. Wang. Synaptic mechanisms and network dynamics underlying spatial working memory in a cortical network model. *Cerebral Cortex*, 10(9):910, 2000.

[CJL+92] M.J. Coster, A. Joux, B.A. LaMacchia, A.M. Odlyzko, C.P. Schnorr, and J. Stern. Improved low-density subset sum algorithms. *Computational Complexity*, 2(2):111–128, 1992.

[CK09] J.C. Curtis and D. Kleinfeld. Phase-to-rate transformations encode touch in cortical neurons of a scanning sensorimotor system. *Nature neuroscience*, 12(4):492–501, 2009.

[CR07] A. Carnell and D. Richardson. Parallel computation in spiking neural nets. *Theoretical Computer Science*, 386(1-2):57–72, 2007.

[CUKH96] S.M. Courtney, L.G. Ungerleider, K. Keil, and J.V. Haxby. Object and spatial visual working memory activate separate neural systems in human cortex. *Cerebral Cortex*, 6(1):39–49, 1996.

[DGA99] M. Diesmann, M.O. Gewaltig, and A. Aertsen. Stable propagation of synchronous spiking in cortical neural networks. *Nature*, 402(6761):529–533, 1999.

[DH11] D. Durstewitz and J. Hass. Models of dopaminergic modulation. *Scholarpedia*, 6(8):4215, 2011.

[DR09] A. Dagher and T.W. Robbins. Personality, addiction, dopamine: insights from Parkinson's disease. *Neuron*, 61(4):502–510, 2009.

[DSS00] D. Durstewitz, J.K. Seamans, and T.J. Sejnowski. Neurocomputational models of working memory. *Nature Neuroscience*, 3:1184–1191, 2000.

[FBGR89] S. Funahashi, C.J. Bruce, and P.S. Goldman-Rakic. Mnemonic coding of visual space in the monkey's dorsolateral prefrontal cortex. *Journal of Neurophysiology*, 61(2):331, 1989.

[Fus73] J.M. Fuster. Unit activity in prefrontal cortex during delayed-response performance: Neuronal correlates of transient memory. *Journal of Neurophysiology*, 1973.

[GH93] S.E. Gathercole and G.J. Hitch. Developmental changes in short-term memory: A revised working memory perspective. *Theories of memory*, 1:189–209, 1993.

[GK02] W. Gerstner and W.M. Kistler. Mathematical formulations of Hebbian learning. *Biological Cybernetics*, 87(5):404–415, 2002.

[GMWK07] O. George, C.D. Mandyam, S. Wee, and G.F. Koob. Extended access to cocaine self-administration produces long-lasting prefrontal cortex-dependent working memory impairments. *Neuropsychopharmacology*, 33(10):2474–2482, 2007.

[GPAW04] S.E. Gathercole, S.J. Pickering, B. Ambridge, and H. Wearing. The structure of working memory from 4 to 15 years of age. *Developmental psychology*, 40(2):177, 2004.

[Heb49] D.O. Hebb. *The Organization of Behavior*. Wiley, New York, 1949.

[HM96] A.F. Healy and D.S. McNamara. Verbal learning and memory: Does the modal model still work? *Annual Review of Psychology*, 47(1):143–172, 1996.

[HT09] S.A. Harrison and F. Tong. Decoding reveals the contents of visual working memory in early visual areas. *Nature*, 458(7238):632–635, 2009.

[ID03] E.M. Izhikevich and N.S. Desai. Relating STDP to BCM. *Neural Computation*, 15(7):1511–1523, 2003.

[Izh06] E.M. Izhikevich. Polychronization: Computation with spikes. *Neural Computation*, 18(2):245–282, 2006.

[Izh07] Eugene M. Izhikevich. Solving the distal reward problem through linkage of STDP and dopamine signaling. *Cerebral Cortex*, 17(10):2443–2452, 2007.

[JN37] C.F. Jacobsen and H.W. Nissen. Studies of cerebral function in primates. IV. The effects of frontal lobe lesions on the delayed alternation habit in monkeys. *Journal of Comparative Psychology*, 23(1):101, 1937.

[KOW+97] J. Karhunen, E. Oja, L. Wang, R. Vigario, and J. Joutsensalo. A class of neural networks for independent component analysis. *IEEE Transactions on Neural Networks*, 8(3):486–504, 1997.

[LI95] J.E. Lisman and MA Idiart. Storage of 7+/-2 short-term memories in oscillatory subcycles. *Science*, 267(5203):1512, 1995.

[LMV06] N. Lemon and D. Manahan-Vaughan. Dopamine D1/D5 receptors gate the acquisition of novel information through hippocampal long-term potentiation and long-term depression. *The Journal of neuroscience*, 26(29):7723–7729, 2006.

[LN98] M. Luciana and C.A. Nelson. The functional emergence of prefrontally-guided working memory systems in four-to eight-year-old children. *Neuropsychologia*, 36(3):273–293, 1998.

[LPM08] R. Legenstein, D. Pecevski, and W. Maass. A learning theory for reward-modulated spike-timing-dependent plasticity with application to biofeedback. *PLoS Computational Biology*, 4(10):e1000180, 2008.

[LSW81] R. Llinas, IZ Steinberg, and K. Walton. Relationship between presynaptic calcium current and postsynaptic potential in squid giant synapse. *Biophysical Journal*, 33(3):323–351, 1981.

[MB01] W. Maass and C.M. Bishop, editors. *Pulsed neural networks*. The MIT Press, 2001.

[MBT08] G. Mongillo, O. Barak, and M. Tsodyks. Synaptic theory of working memory. *Science*, 319(5869):1543, 2008.

[MDG08] A. Morrison, M. Diesmann, and W. Gerstner. Phenomenological models of synaptic plasticity based on spike timing. *Biological Cybernetics*, 98(6):459–478, 2008.

[MHC04] P.R. Montague, S.E. Hyman, and J.D. Cohen. Computational roles for dopamine in behavioural control. *Nature*, 431(7010):760–767, 2004.

[Mil56] G.A. Miller. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review*, 63(2):81, 1956.

[MMB+04] P.R. Montague, S.M. McClure, PR Baldwin, P.E.M. Phillips, E.A. Budygin, G.D. Stuber, M.R. Kilpatrick, and R.M. Wightman. Dynamic gain control of dopamine delivery in freely moving animals. *Journal of Neuroscience*, 24(7):1754, 2004.

[MMS+09] A. Mita, H. Mushiake, K. Shima, Y. Matsuzaka, and J. Tanji. Interval time coding by neurons in the presupplementary and supplementary motor areas. *Nature neuroscience*, 2009.

[MS95] Z.F. Mainen and T.J. Sejnowski. Reliability of spike timing in neocortical neurons. *Science*, 268(5216):1503, 1995.

[Nad09] Zoltan Nadasdy. Information encoding and reconstruction from the phase of action potentials. *Frontiers in Systems Neuroscience*, 3(0), 2009.

[Oja82] E. Oja. Simplified neuron model as a principal component analyzer. *Journal of mathematical biology*, 15(3):267–273, 1982.

[OL96] N.A. Otmakhova and J.E. Lisman. D1/D5 dopamine receptor activation increases the magnitude of early long-term potentiation at ca1 hippocampal synapses. *The Journal of neuroscience*, 16(23):7478, 1996.

[PAF04] A.G. Phillips, S. Ahn, and S.B. Floresco. Magnitude of dopamine release in medial prefrontal cortex predicts accuracy of memory on a delayed response task. *The Journal of neuroscience*, 24(2):547–553, 2004.

[Rak08] Pasko Rakic. Confusing cortical columns. *Proceedings of the National Academy of Sciences*, 105(34):12099–12100, 2008.

[RM02] G. Rainer and E.K. Miller. Timecourse of object-related neural activity in the primate prefrontal cortex during a short-term memory task. *European Journal of Neuroscience*, 15(7):1244–1254, 2002.

[RPPD05] M. Rudolph, J.G. Pelletier, D. Pare, and A. Destexhe. Characterization of synaptic conductances and integrative properties during electrically induced EEG-activated states in neocortical neurons in vivo. *Journal of neurophysiology*, 94(4):2805, 2005.

[SB98] D.L. Schacter and R.L. Buckner. Priming and the brain. Review. *Neuron*, 20:185–195, 1998.

[Sch07a] W. Schultz. Behavioral dopamine signals. *Trends in Neurosciences*, 30(5):203–210, 2007.

[Sch07b] W. Schultz. Reward signals. *Scholarpedia*, 2(6):2184, 2007.

[SG10] J. Sjöström and W. Gerstner. Spike-timing dependent plasticity. *Scholarpedia*, 5(2):1362, 2010.

[Sho07] H. Z. Shouval. Models of synaptic plasticity. *Scholarpedia*, 2(7):1605, 2007.

[SI10] B. Szatmáry and E.M. Izhikevich. Spike-timing theory of working memory. *PLoS Computational Biology*, 6(8):e1000879, 2010.

[SJT10] C. Savin, P. Joshi, and J. Triesch. Independent component analysis in spiking neurons. *PLoS Computational Biology*, 6(4):e1000757, 2010.

[SLF+09] K. Sidiropoulou, F.M. Lu, M.A. Fowler, R. Xiao, C. Phillips, E.D. Ozkan, M.X. Zhu, F.J. White, and D.C. Cooper. Dopamine modulates an mGluR5-mediated depolarization underlying prefrontal persistent activity. *Nature neuroscience*, 12(2):190–199, 2009.

[TNC09] M.T. Todd, Y. Niv, and J.D. Cohen. Learning to use working memory in partially observable environments through dopaminergic reinforcement. In *Advances in Neural Information Processing Systems 21*, pages 1689–1696, 2009.

[Tri05] J. Triesch. A gradient rule for the plasticity of a neuron's intrinsic excitability. *Int. Conf. on Artificial Neural Networks (ICANN 2005)*, pages 65–70, 2005.

[Tri07] J. Triesch. Synergies between intrinsic and synaptic plasticity mechanisms. *Neural Computation*, 19(4):885–909, 2007.

[UCH98] L.G. Ungerleider, S.M. Courtney, and J.V. Haxby. A neural system for human visual working memory. *Proceedings of the National Academy of Sciences*, 95(3):883, 1998.

[US09] R. Urbanczik and W. Senn. Reinforcement learning in populations of spiking neurons. *Nature Neuroscience*, 12(3):250–252, 2009.

[VWVHW04] I. Van Welie, J.A. Van Hooft, and W.J. Wadman. Homeostatic scaling of neuronal excitability by synaptic modulation of somatic hyperpolarization-activated Ih channels. *Proceedings of the National Academy of Sciences of the United States of America*, 101(14):5123, 2004.

[WKH97] M. Watanabe, T. Kodama, and K. Hikosaka. Increase of extracellular dopamine in primate prefrontal cortex during a working memory task. *Journal of Neurophysiology*, 78(5):2795–2798, 1997.

6 List of Figures

7 List of Tables

8 Appendix

8.1 Additional graphs

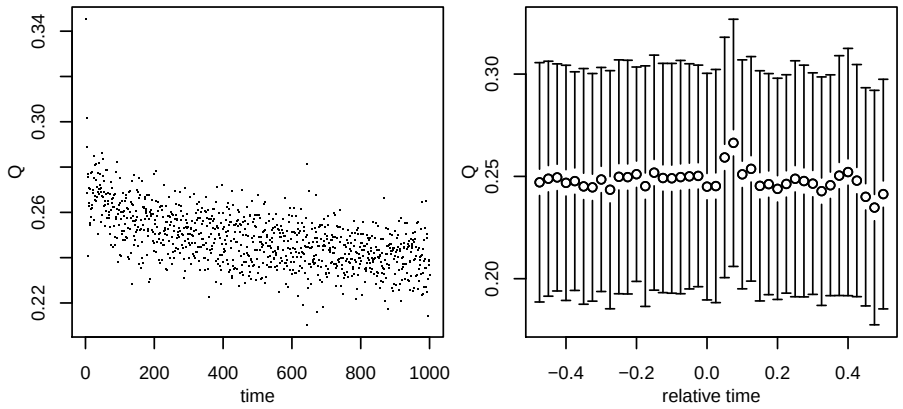


Figure 8.1: Time dependency of the fraction of 0-egal spikes (Q). *Left:* Q during $1000\,s$ simulation, measured over $1\,s$ time bins. *Right:* Mean and standard deviation of the time course of Q during the 692 training trials of one $1000\,s$-simulation with $25\,ms$ time bins. Time is relative to the onset of input stimulation ($t = 0$ in this graph).

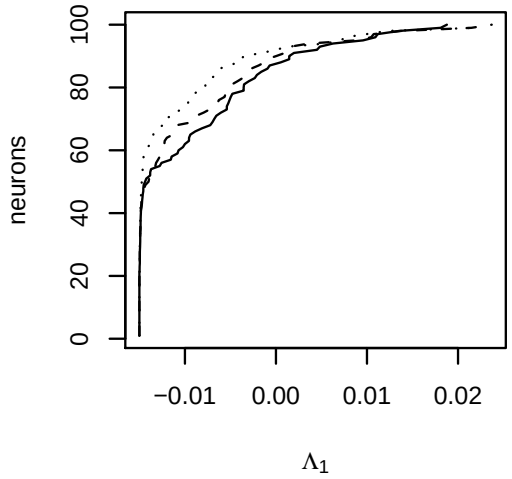


Figure 8.2: Cumulative $\Lambda_{1,i}$ distribution of 100 random, excitatory neurons after $1000\,s$ simulation for three values of τ_u: $50\,ms$ (solid), $100\,ms$ (dashed), $200\,ms$ (dotted).

8.2 RaSimu model description language

The RaSimu software developed within this work uses a domain specific language (DSL) to allow the rapid formulation of hybrid differential delay equation systems. These descriptions are then compiled into model-specific programs responsible for simulation and analysis of the model.

A complete model specification has to declare each model variable, including a name, a type (e.g. `double`) and site (e.g. neuron or spike). If the variable is continuous (belongs to the global, neuron or synapse site; compare with section 8.4), an initial value has to be specified. For temporal evolution and reacting on events, update rules have to be written. If they are omitted, the default rules—constant over time and no change on events—apply. The syntax for update rule and declaration is:

`declaration` := *type name* [= *default*];
`update-rule` := *name'* = *expression*;

Each variable belongs to exactly one site of the simulation: `Global`, `Neuron`, `Synapse, GlobalMsg`, `Spike`, `RandomSpike` or `SpikeArrival`. The expression in the update rule is interpreted as C++, except that it can use any variable of the model, as long as the instance to be accessed is unambiguous: For example a synapse variable may access a neuron variable of the neuron it belongs to[1]. The other way around is not possible in general, because each neuron has several synapses. The only exception from this rule is during `SpikeArrival` events: in their case variables of the affected neuron *and* synapse may be used. If a tick is appended to a variable—like `Voltage'`—this refers to its value after temporal evolution, event application or post-event condition (depending on the context). Otherwise it refers to the prior value. Circular dependencies lead to a compiler crash :-)

Each site is described using one or more `quant-block`s of the following syntax:

`quant-block` :=
<discrete|continuous> *site* { `on-block`∗ `emit-block`∗ `update-rule`∗ }
`on-block` := on *event* { `update-rule`∗ }
`emit-block` := emit *event* { *condition* `update-rule`∗ }

Using the `on-block`, one can describe to which events a site is sensitive (for example a `Neuron` shall receive `RandomSpike` events) and how variables of this site are changed due to the event. Using the `emit-block`, one can define the condition under which a site emits an event, with an optional delay. In this case the update rules define the value of discrete variables of the event to be generated and the post-event condition of the site's variables.

The program to parse and transform this DSL into a C++ fragment is shown in Appendix 8.5.1. It includes the complete, human-readable grammar—for brevity, several small features have been omitted in the preceding explanation.

[1] Per definition synapses belong to the postsynaptic neuron

8.3 Model implementation

```
const const {
    double FireThreshold = 0.015;     // [V]
    double Tau_Voltage = 0.05;        // [s]
    double RefractoryPeriod = 0.001; // [s]
    double Tau_Dopamine = 0.005;      // [s]
    double TauEligibility = 0.1;      // [s]
    double Tau_LTP = 0.014;           // [s]
    double Tau_LTD = 0.034;           // [s]
    double Delta_LTP = 1;             // [V]
    double Delta_LTD = 1;             // [V]
    double DeltaET_LTP = 0.000103;
    double DeltaET_LTD = 0.000055;

    double MaxWeight = 0.004;         // [V]
    double TargetFreq = 10;           // [Hz]
    double LambdaIP1 = 0.00005;
    double RandomFreq = 3.15;         // [Hz]
    double RandomSpikeWeight = 0.01; // [V]
}

discrete GlobalMsg {
    TrainerT NextTrainer;
}

continuous Global {
    double Dopamine = 0.0;
    TrainerT Trainer = TrainerT();
    bool ResetSpikeCounter = true;

    Dopamine' = Dopamine * exp(-dt / Tau_Dopamine);

    on GlobalMsg {
        Trainer'           = NextTrainer;
        ResetSpikeCounter' = NextTrainer.resetCounter;
    }

    emit GlobalMsg {
        default true;
        after Trainer'.delay;

        Dopamine'    = Dopamine + Trainer.reward;
        NextTrainer' = Trainer.update(pc, indices, queues, t);
    }
}
```

```
continuous Neuron {
    double Voltage = 0.0;
    double LTDTrace = 0.0;
    double RefractoryLeft = 0.0;
    double IPCoeff1 = 0.0;
    double SumWeight = 0.0;          // external init
    double TargetSumWeight = 0.0; // external init
    Time LastSpike = 0.0;
    uint16_t SpikeCounter = 0;
    RNG::seed_t RandomSeed = 0;    // external init
    bool RandomEnabled = false;

    Voltage' = Voltage * exp(-dt / Tau_Voltage);
    LTDTrace' = LTDTrace * exp(-dt / Tau_LTD);
    RefractoryLeft' = fmax(0.0, RefractoryLeft - dt);
    Moment1' =  Moment1 * exp(-dt / Tau_MomEst);
    SpikeCounter' = ResetSpikeCounter ? 0 : SpikeCounter;

    on SpikeArrival {
        SumWeight' = SumWeight + DeltaWeight;
        Voltage' = Voltage + Weight;
    }

    on RandomSpike {
        Voltage' = Voltage + RandomSpikeWeight;
        RandomEnabled' = not isExcitatory();
    }

    emit Spike {
        default true;
        if Voltage' > FireThreshold + _CP(IPCoeff1);
        if RefractoryLeft' == 0.0;

        LTDTrace' = Delta_LTD;
        RefractoryLeft' = RefractoryPeriod * isExcitatory()
        Voltage' = 0;
        LastSpike' = t;
        IPCoeff1' = isExcitatory()
          ? fmax(-FireThreshold, IPCoeff1 - (t - LastSpike)()
          * LambdaIP1 * (TargetFreq - 1 / (t - LastSpike)()))
          : IPCoeff1;
        SpikeCounter' = ResetSpikeCounter ? 0 : SpikeCounter + 1;
    }

    emit RandomSpike {
        default false;
        if RandomEnabled' == true;
```

```
        after RNG::expo(RandomSeed, 1.0 / RandomFreq);

        RandomEnabled' = false;
        RandomSeed' = RNG::next(RandomSeed);
    }
}

continuous Synapse {
    double Weight = 0.0;            // external init
    double DeltaWeight = 0;
    double TmpDeltaWeight = 0;
    double EligibilityTrace = 0.0;
    double LTPTrace = 0.0;

    EligibilityTrace' =
        EligibilityTrace * exp(-dt / TauEligibility)
        + (LastSpike == t) * DeltaET_LTP * LTPTrace';
    LTPTrace' = LTPTrace * exp(-dt / Tau_LTP);
    TmpDeltaWeight' = isVariable()
        ? TmpDeltaWeight
          + EligibilityTrace * Dopamine
          * Tau_Dopamine * TauEligibility
          / (Tau_Dopamine + TauEligibility)
          * (1.0 - exp( (-(Tau_Dopamine + TauEligibility))
                        / (Tau_Dopamine * TauEligibility)
                        * dt))
        : 0;

    on Spike {
        LTPTrace' = LTPTrace + Delta_LTP;
        EligibilityTrace' = EligibilityTrace
                          - DeltaET_LTD * LTDTrace;
    }

    emit SpikeArrival {
        default true;

        Weight' = isVariable()
            ? fmin(MaxWeight, fmax(0,
                (Weight + TmpDeltaWeight)
                / (SumWeight + TmpDeltaWeight)
                * TargetSumWeight))
            : Weight;
        DeltaWeight' = Weight' - Weight;
        TmpDeltaWeight' = 0;
    }
}
```

8.4 RaSimu Design Document

RaSimu is a simulator for spiking neural networks implemented primarily in C++11. It uses the event-driven simulation approach [BRC+07]. This section briefly describes noteworthy deviations from the common way to implement such a simulator.

8.4.1 Pretext

Before deciding to implement RaSimu, experience with a number of different approaches to simulate the neural networks described within this thesis has been made:

- using Matlab, based on the model from [Izh07],
- using NEST, augmented with a DA-modulated STDP synapse,
- using custom software written in C++, based around the idea to decouple simulator, trainer and analysis tools as separate UNIX processes, interconnected by pipes using a human-readable data format (CSV).

During that experience several shortcomings surfaced (not all points apply to every approach):

- Model dynamics scattered across several source code files and intermingled with model-irrelevant implementation details increased the error frequency and the debugging effort[2].
- Available tools were not sufficient to rapidly execute new analysis queries because either insufficient data availability required a time-consuming rerun of the simulation with new logging statements, or the amount of anticipatorily logged data was so big that the simulation was unusably slow (and consumed to much storage space).
- Indeterministic simulations—where that output of PRNGs depends on the execution environment—impeded repeatable simulations.
- Simulations where either fast or suitable for rapid prototyping.

This led to the creation of the RaSimu simulator, which alleviates all those shortcoming in exchange for its own significant set of deficiencies. Nonetheless it has to be acknowledged that implementing a custom neural network simulator for a project of the size of this thesis is (time-)economically unjustifiable.

8.4.2 Requirements

The requirements for RaSimu were:

1. Allow a dense formulation of the model in a single file, in a language close to the hybrid differential delay equation systems used in section 2.1.

[2] In addition the model complexity itself covers subtle bugs easily. It has happened to the author more than once, that the homeostatic mechanisms counteracted a low-level bug in the simulator code.

2. Decouple simulation and analysis: Record enough data during simulation to later resume it and to allow replaying temporal and spatial subsets of the simulation with arbitrary precision efficiently.

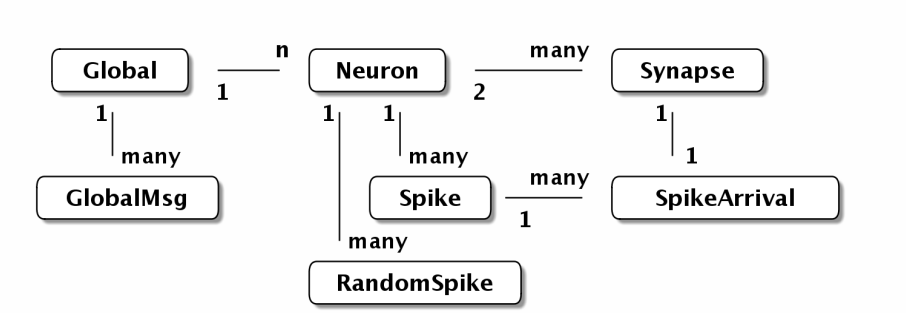
3. Be completely deterministic.

4. Be fast enough.

8.4.3 Data and computation structure

The simulation has an open number of properties. A property's value is derived from its own and other properties' values of the past using a fixed rule.

Ontological Association Each property has an ontological associated type. It is instanced for every object of the given type. The type determines further, under which circumstances the update rule of the property is called and when snapshots of the properties' current values are stored. The available ontological types and their relations are:



The number of instances for each ontological type, as well as the relation between instances of different type is compile-time fixed—except for the network topology.

Temporal nature All properties of the first row of the above graph are continuous, all types of second and third row discrete.

For discrete properties, every single event is stored. Continuous properties are only stored at checkpoints. Checkpoints are roughly at regular time intervals but fine structured to exactly match the time of incoming events, so that a continuous property is only stored at moments of modification by an discrete property. This allows to replay the simulation (or parts of it) exactly, to the level of floating-point instruction order. Discrete properties are indexed, so that the set of spikes from and to a certain neuron in a certain time interval can be efficiently determined. Indices and checkpoints together allow to exactly replay selected parts of the simulation with constant time overhead.

8.4.4 Implementation

A few implementation-approaches taken to meet the requirement are noteworthy:

Memory-mapped data structures All data structures are stored in memory-mapped files. This allows multiple program instances (e.g. different analysis tasks) to run in parallel and share memory-intensive data, instead of competing for memory.

Row-wise data layout The data storage for each property is laid out continuously in memory—instead of storing all properties of one instance and one time closely together. This allows to only load those properties into memory that are strictly required for the current task. This reduces the pressure on the CPU cache.

Compile-, link-, and run-time assertions The programs are build to fail as early as possible. The type system is used to prevent accidental assignment of instance pointers of different ontological type, for example. Link-time assertions (code that fails to link if the compiler is unable to optimize it away) detect circular dependencies and some illegal data access patterns. Finally, the code is littered with run-time assertion.

Meta-programming Compile-time template meta-programming and the described domain specific language are used to automatically construct the necessary data structures for simulation and several functions, to operate on them, during compilation. This allows a compact formulation of the simulation kernel (see below) and a fast simulation despite a versatile model description language, due to compile-time optimizations.

8.5 Source code excerpts

8.5.1 DSL conversion

The following Perl6-code translates the model description from the DSL into C++-code, that is included in all model dependent parts of RaSimu.

```
1  #!/usr/bin/perl6
2  use v6;
3
4  grammar DSL {
5      my $e = "document start";
6      token TOP { ^ <include_block>* <quant_block>* $ }
7
8      token include_block {
9      'include' \s+ '"' <filename> '"' \s* ';' \s*
10     }
11     token quant_block {
12     <quant_type> \s+ <name> \s* '{' \s*
13           [[{ $e=$/} <decl>|<evolve>|<on_block>|<emit_block>|<comment>] \
                 s*] *
14     \s* '}' \s* }
15     token on_block {
16     'on' \s+ <name> \s* '{' \s*
17         [[{ $e=$/} <evolve>|<comment>] \s*] *
18     \s* '}' }
19     token emit_block {
```

```
20      'emit' \s+ <name> \s* '{' \s*
21          [[{$e=$/} <evolve>|<default>|<after>|<if_>|<comment>] \s*] *
22      \s* '}' }
23
24      token decl { <type> \s+ <name> \s* ['=' \s* <expr> \s*]? ';' }
25      token evolve { <name>\' \s* '=' \s* <expr> \s* ';' }
26      token default { 'default' \s+ <bool> \s* ';' }
27      token if_ { 'if' \s+ <name>\' <expr> \s* ';' }
28      token after { 'after' \s+ <expr> \s* ';' }
29      token comment { '//' \N* $$ }
30
31      token name { \w+ }
32      token filename { \w+ }
33      token type { [':'|\w]+ }
34      token quant_type { ['discrete'|'continuous'|'const'] }
35      token bool { ['true'|'false'] }
36      token expr { <-[;{}]>* } # MAYBE TODO: improve
37
38      method error() {
39      return $e;
40      }
41  }
42
43  sub transform($quant_blocks) {
44      my %quant_types;
45      my %vars;
46      my %consts;
47      my %res = :const(''), :decl(''), :evolve(''), :on(''),
48      :qdq(''), :emit(''), :gpe(''), :gen(''), :debug('');
49
50      my %rawexpr_evolve;
51
52      # funs to replace def strings to C++-Code
53      sub replace_consts($expr is copy) {
54          for %consts.kv -> $name, $val {
55          $expr = $expr.subst(rx/<<$name>>/, "$val /* $name */", :g);
56          }
57          return $expr;
58      }
59
60      sub replace_dt($expr is copy) {
61          $expr = $expr.subst(rx/<<'dt'>>/, "td()", :g);
62          return $expr;
63      }
64
65      sub replace_expr($expr is copy, %local_exprs) {
66          # replace post-event vars ONCE
67          for %vars.kv -> $prop, $quant {
68          # post event is only defined for continuous quants
69          next if %quant_types{$quant} eq 'discrete';
70          $expr = $expr.subst(rx/<<$prop>>"'"/, "(%local_exprs{$prop})",
                :g);
71          }
72          return $expr;
73      }
74
75      sub replace_vars($expr is copy) {
```

```
76          # replace normal vars
77          for %vars.kv -> $prop, $quant {
78          if %quant_types{$quant} eq 'continuous' {
79              $expr = $expr.subst(rx/<<$prop>>/, "_CP($prop)", :g);
80          } else {
81              die unless %quant_types{$quant} eq 'discrete';
82              $expr = $expr.subst(rx/<<$prop>>/, "_DP($prop)", :g);
83          }
84          }
85          return $expr;
86      }
87
88      sub replace_all($expr is copy, %local_exprs) {
89          $expr = replace_consts(
90              replace_dt(
91              replace_vars(
92              replace_expr($expr, %local_exprs))));
93          $expr = $expr.subst(rx/"_PROT">>/, "q", :g);
94          return $expr;
95      }
96
97      sub replace_all_after($expr is copy, %local_exprs) {
98          $expr = $expr.subst(rx/"'"/, "_PROTTICK", :g);
99          $expr = replace_vars($expr);
100         $expr = $expr.subst(rx/"_PROTTICK"/, "'", :g);
101         return replace_consts(
102            replace_expr($expr, %local_exprs));
103     }
104
105     # gather all properties (required to patch expressions)
106     for @($quant_blocks) {
107         my $qc = $_<name>;
108         my $qt = $_<quant_type>;
109         %quant_types{$qc} = $qt;
110         for @($_<decl>) {
111         my $name = $_<name>;
112         if $qt eq 'const' {
113             if not %consts{$name} {
114             my $type = $_<type>;
115             %consts{$name} = $_<expr>;
116             %res<const> ~= "const $type $name = $_<expr>;\n";
117             }
118         } else {
119             die "multiple declaration of $_<name>" if %(%vars, %consts)
                    {$name};
120             %vars{$name} = $qc;
121         }
122         }
123         for @($_<evolve>) { %rawexpr_evolve{$_<name>} = $_<expr>; }
124     }
125
126     # print definitions
127     for @($quant_blocks) {
128         my $qc = $_<name>;
129         my $emit = $_<emit_block>;
130         my %types;
131         next if ($qc eq 'const');
```

```
132         %res = %res <<~>> "// Quantor: $qc\n";
133         for @($_<decl>) {
134         %types{$_<name>} = $_<type>;
135         if %quant_types{%vars{$_<name>}} eq 'continuous' {
136             %res<decl> ~=
137             "GEN_CP($qc, $_<name>, \"$_<name>\", $_<type>, ("
138             ~ replace_consts($_<expr>) ~ "));\n";
139         } else {
140             %res<decl> ~=
141             "GEN_DP($qc, $_<name>, \"$_<name>\", $_<type>);\n";
142         }
143         }
144         for @($_<evolve>) {
145         %res<evolve> ~= "GEN_CP_EVOLVE($_<name>, "
146             ~ replace_all($_<expr>, %rawexpr_evolve) ~ ");\n";
147         }
148         for @($_<emit_block>) {
149         my $dqc = $_<name>;
150         my %rawexpr_emit =
151             gather for @($_<evolve>) { take %($_<name>, $_<expr>); };
152         my %rawexpr_after =
153             gather for @(%vars) { my $k=$_.key; take %($k, "_TP($k)");
                };
154
155         %res<emit> ~= "GEN_QUANT_EMIT($qc, $dqc, $_<default>[0]<bool>)
            ;\n";
156         if $_<after>.elems > 0 {
157             %res<emit> ~=
158             "GEN_QUANT_HASVARDELAY($dqc);\n"
159             ~ "GEN_DP_DELAY($dqc, $qc, "
160             ~ replace_all_after($_<after>[0]<expr>, %rawexpr_after)
161             ~ ");\n";
162         }
163         for @($_<evolve>) {
164             die "$_<name> of %vars{$_<name>} != $qc, $dqc"
165             unless (%vars{$_<name>} eq $dqc) || (%vars{$_<name>} eq $qc
                );
166             %res<gpe> ~=
167             "GEN_CP_GENERATE($qc, $dqc, $_<name>) \{ "
168             ~ "value = " ~ replace_all($_<expr>, %rawexpr_emit)
169             ~ "; \}\};\n";
170         }
171         }
172         for @($_<on_block>) {
173         my $src_quant = $_<name>;
174         %res<qdq> ~= "GEN_QDQ($_<name>, $qc);\n";
175         my %is_evolved;
176         my %rawexpr_on =
177             gather for @($_<evolve>) { take %($_<name>, $_<expr>); };
178         for @($_<evolve>) {
179             my $dprop = $_<name>;
180             %res<on> ~= "GEN_CP_APPLY($dprop, $src_quant, true) \{\n"
181             ~ "  const %types{$dprop} tmp = "
182             ~ replace_all($_<expr>, %rawexpr_on) ~ ";\n"
183             ~ "  transaction.template set<$dprop>(tmp);\n";
184             for @($emit) {
185             my $dst_quant = $_<name>;
```

```
186             my %rawexpr_emit =
187                 gather for @($_<evolve>) { take %($_<name>, $_<expr>);
                        };
188             for @($_<if_>) {
189                 if $_<name> eq $dprop {
190                 %res<on> ~= "  intent.template get<$dst_quant>()"
191                     ~ ".combine(tmp" ~ replace_consts($_<expr>)
192                     ~ ");\n";
193                 }
194             }
195             }
196             %res<on> ~= "\}\};\n"; # sic!
197             %is_evolved{$dprop} = True;
198         }
199         for @($emit) {
200             my $dst_quant = $_<name>;
201             for @($_<if_>) {
202             unless %is_evolved{$_<name>} {
203                 # HINT: use _CP for access because the value is
                        unchanged
204                 %res<on> ~=
205                 "GEN_CP_APPLY($_<name>, $src_quant, false) \{\n"
206                 ~ "  intent.template get<$dst_quant>()"
207                 ~ ".combine(_CP($_<name>)" ~ replace_consts($_<expr>)
208                 ~ ");\n\}\};\n";
209             }
210             }
211         }
212         }
213     }
214
215     # create list of
216     %res<lists> =
217         "typedef boost::mpl::list<\n\t"
218         ~ %vars.keys.map({"boost::mpl::pair<$_, boost::mpl::bool_<true
              >>"}).join(",\n\t")
219         ~ "\n> all;\n\n"
220         ~ "typedef boost::mpl::list<\n\t"
221         ~ %vars.keys.map({"boost::mpl::pair<$_, boost::mpl::bool_<false
              >>"}).join(",\n\t")
222         ~ "\n> all_ro;";
223
224     # push results
225     %res = %res <<~>> "\n";
226     say "#ifndef RASIMU_SIMULATION_DEFINITION
227 #define RASIMU_SIMULATION_DEFINITION
228
229 #include \"core/property_abbrevations_begin.hpp\"
230
231 namespace ModelConsts \{
232 %res<const>
233 \}
234 %res<decl>
235 %res<evolve>
236 %res<on>
237 %res<qdq>
238 %res<emit>
```

```
239 %res<gen>
240 %res<gpe>","
241
242 namespace Lists \{
243 %res<lists>","
244 \}
245
246 // generated by ",qqx{perl6 --version|grep version},"
247
248 /* DEBUG MSGS
249 %res<debug>
250 */
251 #include \"core/property_abbrevations_end.hpp\"
252 #endif // RASIMU_SIMULATION_DEFINITION";
253 }
254
255 sub parse($src) {
256     my $p = DSL.new;
257     if $p.parse(slurp $src) {
258     my @qbs = @($<quant_block>);
259     for @($<include_block>) {
260         @qbs.push(parse($_<filename> ~ ".model"));
261     }
262     return @qbs;
263     } else {
264     die "Error after:\n", $p.error;
265     }
266 }
267
268 sub MAIN($src) {
269     transform(parse($src));
270 }
```

8.5.2 Simulation

The following code show the simulation kernel, that generates the data used in later analysis. A lot of C++ boilerplate code has been omitted in this and all following code excerpts.

```
1  template<typename PropList>
2  struct SimLoop {
3    typedef PropertyComposition<PropList> pc_t;
4
5    SimLoop()
6      : indices(),
7        queues()
8    {}
9
10   bool run(const Time until, uint64_t maxEvents) __attribute__((
       noinline)) {
11     Time old(queues.min());
12     while (queues.min() <= until && maxEvents) {
13       assert(old <= queues.min());
14       old = queues.min();
15       switch (queues.minType()) {
16 #define HT(T) case RuntimeID<T>::value: handleEvent<T>(); break;
```

```
17      HT(SpikeArrival);
18      HT(Spike);
19      HT(RandomSpike);
20      HT(GlobalMsg);
21  #undef HT
22        default: assert(false);
23        }
24        maxEvents--;
25      }
26      return maxEvents;
27    }
28
29    template<typename Quant>
30    inline void handleEvent() {
31      typedef typename QuantDestinationQuant<Quant>::type DstQuant;
32      Time ct(queues.min());
33      Event<Quant> &event = queues.get<Quant>()(ct).minPayload();
34      Ptr<DstQuant> dst(event.dst(topology));
35
36      // deliever the event and check if to create new events
37      DelieverContext<Quant, DstQuant> delieverContext(event.instance(),
            dst);
38      PLA_Apply<Quant, DstQuant, pc_t> apply(ct, delieverContext);
39      pc.call(apply);
40
41      // if we create any events
42      if (apply.generateAny()) {
43        // evolve all entities which depend on our pre-event values
44        Evolve<DstQuant>()(pc, ct, dst);
45
46        // compute delay values (they require access to the pre- and
47        // post-event values and can thus only be computed here)
48        apply.computeDelay(pc);
49      }
50
51      // commit data calculated during apply
52      //   this is necessary because childs are only eventually evolved,
53      //   but require the pre-event values of the instance we are
54      //   committing to now
55      apply.commit(pc);
56
57      // generate events and compute discrete properties
58  #define MGE(DQ)                                          \
59      if (apply.template generate<DQ>())                   \
60        generateEvent<DstQuant, DQ, Quant>   (ct, dst, apply);
61
62      MGE(GlobalMsg);
63      MGE(Spike);
64      MGE(RandomSpike);
65  #undef MGE
66      if (apply.template generate<SpikeArrival>())
67        generateSAEvent(ct, event, apply);
68
69      if (boost::is_same<Quant, Spike>::value) {
70        // reinsert spike to get to the next neuron; also handles
71        // removing current min element from the queue (because of a
72        // race condition)
```

```
73        reinsertSpike(ct, FORCE_CONV(Event<Spike>, event));
74      }else{
75        queues.removeLocalMin<Quant>(ct);
76      }
77    }
78
79    template<typename ContQuant, typename EventQuant, typename SrcEvQuant
        >
80    inline void generateEvent(Time ct,
81                  typename ContQuant::instance_ptr_t src,
82                  PLA_Apply<SrcEvQuant, ContQuant, pc_t> &apply) {
83      // generate new discrete quant instance
84      Time eventTime = ct + apply.template getDelay<EventQuant>();
85      Ptr<EventQuant> ptrNE(indices.get<Index<EventQuant>>().add(ct,
            eventTime, src));
86      EmitContext<ContQuant, EventQuant> emitContext(src, ptrNE);
87      PLA_Generate<ContQuant, EventQuant, indices_t, queues_t>
88        pla_gen(ct, emitContext, indices, queues);
89      pc.call(pla_gen);
90
91      // generate new event
92      if (likely((not TopologicalEventDiscard<ContQuant, EventQuant>()(
            topology, src))))
93        queues.insert
94      (ct,
95       eventTime + TopologicalTimeOffset<ContQuant, EventQuant>()(
            topology, src),
96       Event<EventQuant>(eventTime, src, ptrNE));
97    }
98
99    template<typename EventT, typename ContQuant, typename SrcEvQuant>
100   inline void generateSAEvent(Time ct,
101                 EventT &rawEvent,
102                 PLA_Apply<SrcEvQuant, ContQuant, pc_t> &apply) {
103     // garantuee that we are called by spike event and cast alike
104     assert((boost::is_same<EventT, Event<Spike>>::value));
105     assert((boost::is_same<ContQuant, Synapse>::value));
106     Event<Spike> &event(FORCE_CONV(Event<Spike>, rawEvent));
107     Ptr<Synapse> src(event.dst(topology));
108
109     // calculate the SA-ptr
110     Ptr<SpikeArrival> ptrNE(event.instance(), event.offset);
111
112     // generate all SA properties (except voltage difference)
113     EmitContext<Synapse, SpikeArrival> emitContext(src, ptrNE);
114     PLA_Generate<Synapse, SpikeArrival, indices_t, queues_t>
115       pla_gen(ct, emitContext, indices, queues);
116     pc.call(pla_gen);
117
118     // generate event and add to queues
119     queues.insert(ct, ct, Event<SpikeArrival>(src, ptrNE));
120   }
121
122   inline void reinsertSpike(Time ct, Event<Spike> &event) {
123     // TODO (optimize): reuse the current event
124     Event<Spike>::offset_t offset(event.offset + 1);
125
```

```
126       if (unlikely((topology.synapse(event.src, offset)() == Topology::
              nil()()))) {
127         queues.removeLocalMin<Spike>(ct);
128       }else{
129         // create new event (depends on old)
130         Event<Spike> newEvent
131       (event.baseTime,
132        event.src,
133        event.spike,
134        offset);
135
136         // delete old event; this must happen before insertion of the
137         // new event to avoid race in case of equal times
138         queues.removeLocalMin<Spike>(ct);
139
140         // insert new event
141         queues.insert
142       (ct,
143        newEvent.baseTime + topology.time(newEvent.src, offset),
144        newEvent);
145       }
146     }
147
148     void sync() {
149       // TODO: sync all or remove method
150       pc.cast(PLA_Sync{});
151       indices.get<Index<GlobalMsg>>().sync();
152       queues.get<GlobalMsg>().sync();
153     }
154
155     Time currentTime() {
156       return queues.min();
157     }
158
159     /// persistant data storage (holds every mmap'ed data structure)
160
161     // properties
162     pc_t pc;
163     Topology topology;
164
165     // indices
166     template<typename T> struct index_from_type {
167       typedef Index<T> type;
168     };
169     typedef typename boost::mpl::transform<
170       DiscreteQuantorList,
171       index_from_type<boost::mpl::_>
172       >::type indices_list_t;
173     typedef typename UnpackTypeList<TypeSet, indices_list_t >::type
            indices_t;
174     indices_t indices;
175
176     // event queues
177     typedef MultiQueue<
178       Time, Event, DiscreteQuantorList
179       > queues_t;
180     queues_t queues;
```

```
181 };
```

8.5.3 Replay

The replay kernel shown below is used to generate the time-course of model variables during a user-specified spatiotemporal subset of the simulation with arbitrary temporal resolution.

```
1  /// factory defining which discrete quants we have to consider
2
3  template<typename ReplayQuant>
4  struct MultiQueueFactory;
5
6  template<>
7  struct MultiQueueFactory<Global> {
8    typedef list<GlobalMsg> dependencies;
9    typedef MultiQueue<Time, FilterPayload, dependencies, FilterContainer
       > type;
10   typedef IdList<Ptr<Global>::ptr_t> id_list_t;
11   static type instance(id_list_t &, const Time time) {
12     return type(time);
13   }
14 };
15
16 template<>
17 struct MultiQueueFactory<Neuron> {
18   typedef list<GlobalMsg, Spike, RandomSpike, SpikeArrival>
       dependencies;
19   typedef MultiQueue<Time, FilterPayload, dependencies, FilterContainer
       > type;
20   typedef IdList<Ptr<Neuron>::ptr_t> id_list_t;
21   static type instance(id_list_t ids, const Time time) {
22     return type(Filter<GlobalMsg>   (time),
23         Filter<Spike>       (ids, time),
24         Filter<RandomSpike> (ids, time),
25         Filter<SpikeArrival>(ids, time));
26   }
27 };
28
29 template<>
30 struct MultiQueueFactory<Synapse> : MultiQueueFactory<Neuron> {
31   typedef IdList<Ptr<Synapse>::ptr_t> id_list_t;
32   static type instance(id_list_t ids_synapse, Time time) {
33     MultiQueueFactory<Neuron>::id_list_t ids_neuron;
34     for (auto syn : ids_synapse)
35       ids_neuron.insert(Ptr<Synapse>(syn).extractNeuron()());
36     return MultiQueueFactory<Neuron>::instance(ids_neuron, time);
37   }
38 };
39
40 /// template brainfuck to prevent instancing unnecessary handleEvents
     <>()
41
42 template<typename ReplayQuant, typename Quant>
43 struct MaybeHandleEvent {
44   template<typename Q> struct HandleEvent {
```

```
45      template<typename Sim> void operator() (Sim &sim) {
46        sim.template handleEvent<Q>();
47      }
48    };
49
50    struct IgnoreEvent {
51      template<typename Sim> void operator() (Sim &sim) {}
52    };
53
54    typedef typename contains<
55      typename MultiQueueFactory<ReplayQuant>::dependencies,
56      Quant>::type isRelevant;
57    typedef typename if_<isRelevant,
58                  HandleEvent<Quant>,
59                  IgnoreEvent
60                  >::type impl;
61  };
62
63
64  /// extract neurons from arbitrary id list
65  template<typename Quant> struct ExtractNeuronIDs;
66
67  template<>
68  struct ExtractNeuronIDs<Global> {
69    IdList<Ptr<Neuron>::ptr_t> operator() (IdList<Ptr<Global>::ptr_t> &)
         {
70      return IdList<Ptr<Neuron>::ptr_t>();
71    }
72  };
73
74  template<>
75  struct ExtractNeuronIDs<Neuron> {
76    IdList<Ptr<Neuron>::ptr_t> operator() (IdList<Ptr<Neuron>::ptr_t> &
         src) {
77      return src;
78    }
79  };
80
81  template<>
82  struct ExtractNeuronIDs<Synapse> {
83    IdList<Ptr<Neuron>::ptr_t> operator() (IdList<Ptr<Synapse>::ptr_t> &
         src) {
84      IdList<Ptr<Neuron>::ptr_t> res;
85      for (auto id : src)
86        res.insert(Ptr<Synapse>(id).extractNeuron()());
87      return res;
88    }
89  };
90
91  /// the precious replay
92
93  template<typename PropList, typename ReplayQuant,
94       template<class, class> class EventHandler = MaybeHandleEvent>
95  struct SimReplay {
96    typedef PropertyComposition<PropList> PropComp;
97    typedef Checkpoint<Void, 1> Cp;
98    typedef IdList<typename Ptr<ReplayQuant>::ptr_t> id_list_t;
```

```
99    typedef IdList<typename Ptr<Neuron>::ptr_t> neuron_list_t;
100
101   SimReplay(id_list_t ids, Time start)
102     // we need the previous checkpoint interval
103     : ct(Cp::interval() * (std::max((const uint32_t) 1, Cp::binTime(
          start)) - 1)),
104       ids(ids),
105       neurons(ExtractNeuronIDs<ReplayQuant>()(ids)),
106       queues(std::move(MultiQueueFactory<ReplayQuant>::instance(ids, ct
            )))
107   {
108     // init values of our quant
109     pc.cast(PLA_InitByCopy{ct});
110
111     // forward precisely to the request time
112     run(start, -1);
113   }
114
115   template<typename Prop>
116   typename Prop::type get(Ptr<ReplayQuant> id, Time t) {
117     assert(t >= ct);
118     assert(ids.count(id()));
119     run(t, -1);
120     typedef ContinuousContext<ReplayQuant> Ctx;
121     PLA_Get<Prop, Ctx> pla_get(t, Ctx(id));
122     return pc.call(pla_get);
123   }
124
125   bool run(const Time until, uint64_t maxEvents) {
126     assert(until != Time::never());
127     while (queues.min() <= until && maxEvents) {
128       ct = queues.min();
129       switch (queues.minType()) {
130 #define HT(T) case RuntimeID<T>::value: typename EventHandler<
        ReplayQuant, T>::impl()(*this); break;
131     HT(SpikeArrival);
132     HT(Spike);
133     HT(RandomSpike);
134     HT(GlobalMsg);
135 #undef HT
136       default: assert(false);
137       }
138       --maxEvents;
139     }
140     return maxEvents;
141   }
142
143   template<typename Quant>
144   inline bool handleEvent() {
145     typedef typename QuantDestinationQuant<Quant>::type DstQuant;
146     typename CalcDPtr<Quant>::type src(queues.get<Quant>().minPayload()
          .src);
147     Ptr<DstQuant>                  dst(queues.get<Quant>().minPayload()
          .dst);
148
149     // deliever the event and check if to create new events
150     DelieverContext<Quant, DstQuant> delieverContext{src, dst};
```

```
151       PLA_Apply<Quant, DstQuant, PropComp> apply(ct, delieverContext);
152       pc.call(apply);
153
154       // create events
155       if (apply.generateAny()) {
156         // determine how much we have to simulate:
157         if (is_same<ReplayQuant, Global>::value) {
158       // - global replay don't needs any childs
159       pc.cast(PLA_Evolve<Global>{ContinuousContext<Global>{Ptr<Global
              >{}}, ct});
160         }else{
161       if (is_same<DstQuant, Global>::value) {
162         // - neuron/synapse replay needs global and selected neurons
163         for (auto id : neurons)
164           Evolve<Neuron>()(pc, ct, Ptr<Neuron>{id});
165         pc.cast(PLA_Evolve<Global>{ContinuousContext<Global>{Ptr<Global
              >{}}, ct});
166       }else{
167         // - neuron/synapse evolve happens as usual
168         Evolve<DstQuant>()(pc, ct, dst);
169       }
170         }
171       }
172       // commit data calculated during apply
173       //   this is necessary because childs are only eventually evolved,
174       //   but require the pre-event values of the instance we are
175       //   committing to now
176       apply.commit(pc);
177
178       // call the continuous property related post-event triggers to
179       // update state
180   #define HGE(Q) \
181       if (apply.template generate<Q>()) \
182         handleEventGeneration<DstQuant, Q, Quant>(ct, dst, apply);
183
184       HGE(GlobalMsg);
185       HGE(Spike);
186       HGE(RandomSpike);
187       HGE(SpikeArrival);
188   #undef HGE
189
190       queues.template removeMin<Quant>(ct);
191
192       return apply.template generate<Spike>();
193     }
194
195     template<typename ContQuant, typename EventQuant, typename SrcEvQuant
          >
196     inline void handleEventGeneration(Time ct, Ptr<ContQuant> src,
197                       PLA_Apply<SrcEvQuant, ContQuant, PropComp> &)
198     {
199       Void nix;
200       pc.cast(PLA_Generate<ContQuant, EventQuant, Void, Void>
201           (ct, EmitContext<ContQuant, EventQuant>(src, Ptr<EventQuant
                >(-1)),
202            nix, nix));
203     }
```

```
204
205   Time currentTime() {
206     return ct;
207   }
208
209   /// ephermal state
210   Time ct;
211   id_list_t ids;
212   neuron_list_t neurons;
213   typename MultiQueueFactory<ReplayQuant>::type queues;
214   PropComp pc;
215 };
```

8.5.4 egal-Annotation

The code to compute the set of 0-egal spikes arrival is shown below. It is based on the replay kernel above.

```
1  // discrete property Egal
2  struct Egal {
3    typedef uint8_t type;
4    typedef SpikeArrival quant;
5    typedef SpikeArrival::instance_ptr_t instance_ptr_t;
6    static const uint32_t size = 2;
7    static const char* const name;
8  };
9  const char* const Egal::name = "Egal";
10
11 namespace MC = ModelConsts;
12
13 template<typename PropList> struct SimCausality;
14
15 template<typename ReplayQuant, typename Quant>
16 struct MaybeTrackEvent {
17   template<typename Sim>
18   void operator() (Sim &sim) {
19     // we handle every event but only track those affecting neurons
20     auto &sg = static_cast<SimCausality<typename Sim::PropComp::
           properties_t>&>(sim);
21     if (is_same<Quant, SpikeArrival>::value) {
22       sg.handleSA();
23     }else if (is_same<Quant, RandomSpike>::value) {
24       sg.handleRand();
25     }else{
26       sim.template handleEvent<Quant>();
27     }
28   }
29
30   typedef MaybeTrackEvent<ReplayQuant, Quant> impl;
31 };
32
33 template<typename PropList>
34 struct SimCausality : public SimReplay<PropList, Neuron,
       MaybeTrackEvent> {
35   typedef SimReplay<PropList, Neuron, MaybeTrackEvent> Super;
36   using Super::queues;
```

```
37    using Super::pc;
38    using Super::ct;
39    using Super::handleEvent;
40
41    SimCausality(Time start)
42      : Super(IdList<uint16_t>((char*) "0-999"), start),
43        totalMarks{},
44        trueMarks{},
45        inactiveNeurons(1000),
46        hasFired{}
47    {
48      if (start == Time(0)) {
49        // special case: at t=0 all neurons are have zero membrane
50        // voltage and can be used for analysis before firing the first
51        // time
52        inactiveNeurons = 0;
53        for (int i=0; i<1000; i++)
54      hasFired[i] = true;
55      }
56    }
57
58    bool run(const Time until, uint64_t maxEvents) __attribute__((
        noinline)) {
59      return Super::run(until, maxEvents);
60    }
61
62    void handleRand() {
63      Ptr<Neuron> neuron(queues.get<RandomSpike>().minPayload().dst);
64      Ptr<SpikeArrival> fakeSA(Index<SpikeArrival>::nil());
65      handleDiff(neuron, fakeSA, MC::RandomSpikeWeight,
66            &SimCausality<PropList>::template handleEvent<RandomSpike>);
67    }
68
69    void handleSA() {
70      auto &event(queues.get<SpikeArrival>().minPayload());
71      Ptr<SpikeArrival> sa     (event.src.get<0>());
72      Ptr<Synapse>      syn    (event.src.get<1>());
73      Ptr<Neuron>       neuron(event.dst);
74      PLA_Get<Weight> getWeight (ct, syn);
75      Weight::type weight(pc.call(getWeight));
76      handleDiff(neuron, sa, weight,
77            &SimCausality<PropList>::template handleEvent<SpikeArrival>)
                ;
78    }
79
80    void handleDiff(Ptr<Neuron> neuron, Ptr<SpikeArrival> sa, Voltage::
        type diff,
81          bool (SimCausality<PropList>::*handleEvent)()) {
82      PLA_Get<Voltage> getVoltage  (ct, neuron);
83      PLA_Get<IPCoeff1> getIPCoeff1(ct, neuron);
84      Voltage::type  preEventVoltage(pc.call(getVoltage));
85      IPCoeff1::type ipCoeff1       (pc.call(getIPCoeff1));
86      bool evokedSpike = (this->*handleEvent)();
87      auto &openExc(this->openExc[neuron()]);
88      auto &openInh(this->openInh[neuron()]);
89
90      if (!hasFired[neuron()]) {
```

```
91        if (evokedSpike) {
92      hasFired[neuron()] = true;
93      inactiveNeurons--;
94      if (!inactiveNeurons)
95        lastNeuronActivation = ct;
96        }
97        return;
98      }
99
100     if (evokedSpike) {
101       // check exc. list against voltage diff
102       Time wntGap = ct + wntNorm(diff + preEventVoltage
103                 - MC::FireThreshold - ipCoeff1);
104       for (auto i = openExc.lower_bound(wntGap);
105        i != openExc.end();) {
106     mark(i->second, 0);
107     openExc.erase(i++);
108       }
109
110       // mark own spike
111       mark(sa, 0);
112
113       // mark remainig inh/exc spikes as q>0, clear buffers
114       for (auto i : openExc) mark(i.second, 1); openExc.clear();
115       for (auto i : openInh) mark(i.second, 1); openInh.clear();
116     }else{
117       if (diff < 0) {
118     // add to inh. list
119     Time wnt = ct + wntNorm(- diff);
120     openInh.insert(std::make_pair(wnt, sa));
121       }else{
122     // add to exc. list
123     Time wntSelf = ct + wntNorm(diff);
124     openExc.insert(std::make_pair(wntSelf, sa));
125
126     // check inh. list against voltage gap
127     PLA_Get<Voltage> pla_get(ct, neuron);
128     Voltage::type voltage = pc.call(pla_get);
129     Time wntGap = ct + wntNorm(MC::FireThreshold + ipCoeff1 - voltage);
130     for (auto i = openInh.lower_bound(wntGap);
131          i != openInh.end();) {
132       mark(i->second, 0);
133       openInh.erase(i++);
134     }
135       }
136     }
137   }
138
139   void mark(Ptr<SpikeArrival> ptr, bool m) {
140     bool isSA = ptr() != Index<SpikeArrival>::nil();
141     totalMarks[isSA]++;
142     trueMarks[isSA] += !m;
143     if (isSA)
144       egalPC.cast(PLA_Set<Egal>(ptr, 1+m)); // unset values are encoded
              as zero
145   }
146
```

```
147     Time wntNorm(Voltage::type diff) {
148       return MC::Tau_Voltage * log(diff);
149     }
150
151     PropertyComposition<boost::mpl::list<
152       boost::mpl::pair<Egal, boost::mpl::bool_<true>>
153     >> egalPC;
154     uint64_t totalMarks[2],
155       trueMarks[2];
156     uint16_t inactiveNeurons;
157     Time lastNeuronActivation;
158     bool hasFired[maxNeurons];
159     std::multimap<Time, Ptr<SpikeArrival>> openInh[maxNeurons], openExc[
            maxNeurons];
160   };
```

9 Selbstständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.
Seitens des Verfassers bestehen keine Einwände die vorliegende Diplomarbeit für die öffentliche Benutzung im Universitätsarchiv zur Verfügung zu stellen.

Jan Huwald
Jena, 25.11.2011