Chapter 1

Liquid State Machines: Motivation, Theory, and Applications

Wolfgang Maass
Institute for Theoretical Computer Science
Graz University of Technology
A-8010 Graz, Austria
maass@igi.tugraz.at

The Liquid State Machine (LSM) has emerged as a computational model that is more adequate than the Turing machine for describing computations in biological networks of neurons. Characteristic features of this new model are (i) that it is a model for adaptive computational systems, (ii) that it provides a method for employing randomly connected circuits, or even "found" physical objects for meaningful computations, (iii) that it provides a theoretical context where heterogeneous, rather than stereotypical, local gates or processors increase the computational power of a circuit, (iv) that it provides a method for multiplexing different computations (on a common input) within the same circuit. This chapter reviews the motivation for this model, its theoretical background, and current work on implementations of this model in innovative artificial computing devices.

1.1. Introduction

The Liquid State Machine (LSM) had been proposed in [1] as a computational model that is more adequate for modelling computations in cortical microcircuits than traditional models, such as Turing machines or attractor-based models in dynamical systems. In contrast to these other models, the LSM is a model for real-time computations on continuous streams of data (such as spike trains, i.e., sequences of action potentials of neurons that provide external inputs to a cortical microcircuit). In other words: both inputs and outputs of a LSM are streams of data in continuous time. These inputs and outputs are modelled mathematically as functions $u(t)$ and $y(t)$ of continuous time. These functions are usually multi-dimensional (see

Fig. 1.1, Fig. 1.2, and Fig. 1.3), because they typically model spike trains from many external neurons that provide inputs to the circuit, and many different "readouts" that extract output spike trains. Since a LSM maps input streams $u(\cdot)$ onto output streams $y(\cdot)$ (rather than numbers or bits onto numbers or bits), one usually says that it implements a functional or operator (like a filter), although for a mathematician it simply implements a function from and onto objects of a higher type than numbers or bits. A characteristic feature of such higher-type computational processing is that the target value $y(t)$ of the output stream at time t may depend on the values $u(s)$ of the input streams at many (potentially even infinitely many) preceding time points s.

Another fundamental difference between the LSM and other computational models is that the LSM is a model for an *adaptive* computing system. Therefore its characteristic features only become apparent if one considers it in the context of a learning framework. The LSM model is motivated by the hypothesis that the learning capability of an information processing device is its most delicate aspect, and that the availability of sufficiently many training examples is a primary bottleneck for goal-directed (i.e, supervised or reward-based) learning. Therefore its architecture is designed to make the learning as fast and robust as possible. It delegates the primary load of goal-directed learning to a single and seemingly trivial stage: the output- or readout stage (see Fig. 1.4), which typically is a very simple computational component. In models for biological information processing each readout usually consists of just a single neuron, a projection neuron in the terminology of neuroscience, which extracts information from a local microcircuit and projects it to other microcircuits within the same or other brain areas. It can be modelled by a linear gate, a perceptron (i.e., a linear gate with a threshold), by a sigmoidal gate, or by a spiking neuron. The bulk of the LSM (the "Liquid") serves as pre-processor for such readout neuron, which amplifies the range of possible functions of the input streams $u(t)$ that it can learn. Such division of computational processing into Liquid and readout is actually quite efficient, because the same Liquid can serve a large number of different readout neurons, that each learn to extract a different "summary" of information from the same Liquid. The need for extracting different summaries of information from a cortical microcircuit arises from different computational goals (such as the movement direction of objects versus the identity of objects in the case where $u(t)$ represents visual inputs) of different projection targets of the projection neurons. Data from neurophysiology show in fact that for natural stimuli the spike trains of

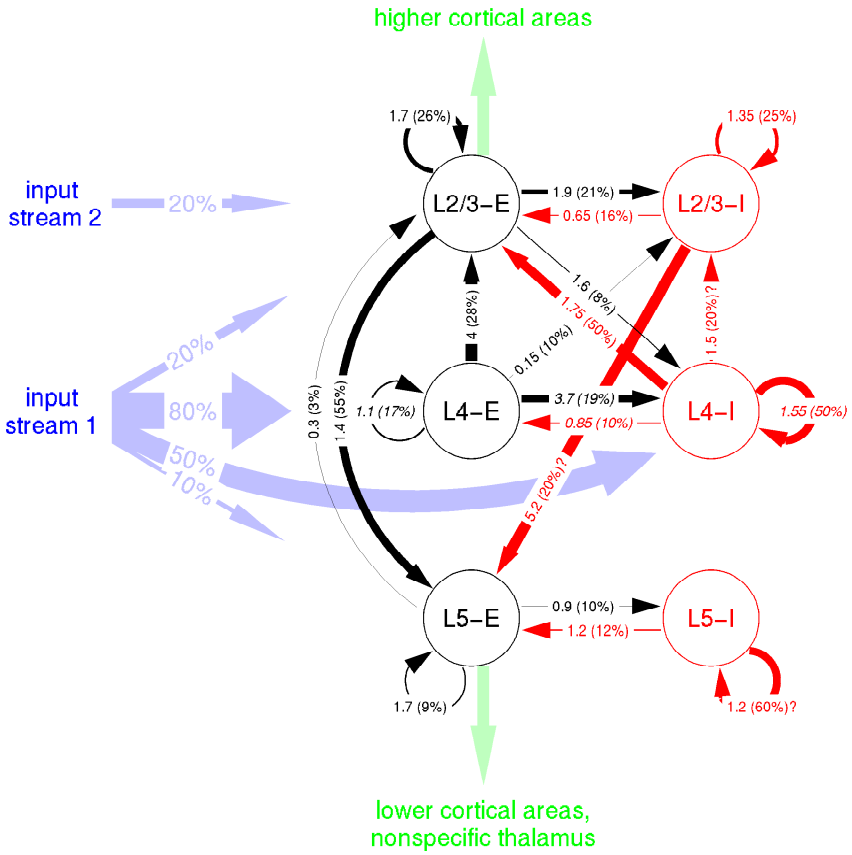

Fig. 1.1.: Modelling a generic cortical microcircuit by a LSM. Template for a generic cortical microcircuit based on data from [2], see [3, 4] for details. The width of arrows indicates the product of connection probabilities and average strength (i.e., synaptic weight) between excitatory (left hand side) and inhibitory (right hand side) neurons on three cortical layers. Input stream 1 represents sensory inputs, input stream 2 represents inputs from other cortical areas. Arrows towards the top and towards the bottom indicate connections of projection neurons ("readouts") on layer 2/3 and layer 5 to other cortical microcircuits. In general these projection neurons also send axonal branches (collaterals) back into the circuit.

different projection neurons from the same column tend to be only weakly correlated. Thus the LSM is a model for multiplexing diverse computations on a common input stream $u(t)$ (see Fig. 1.1, Fig. 1.2, and Fig. 1.3).

One assumes that the Liquid is not adapted for a single computational task (i.e., for a single readout neuron), but provides computational prepro-

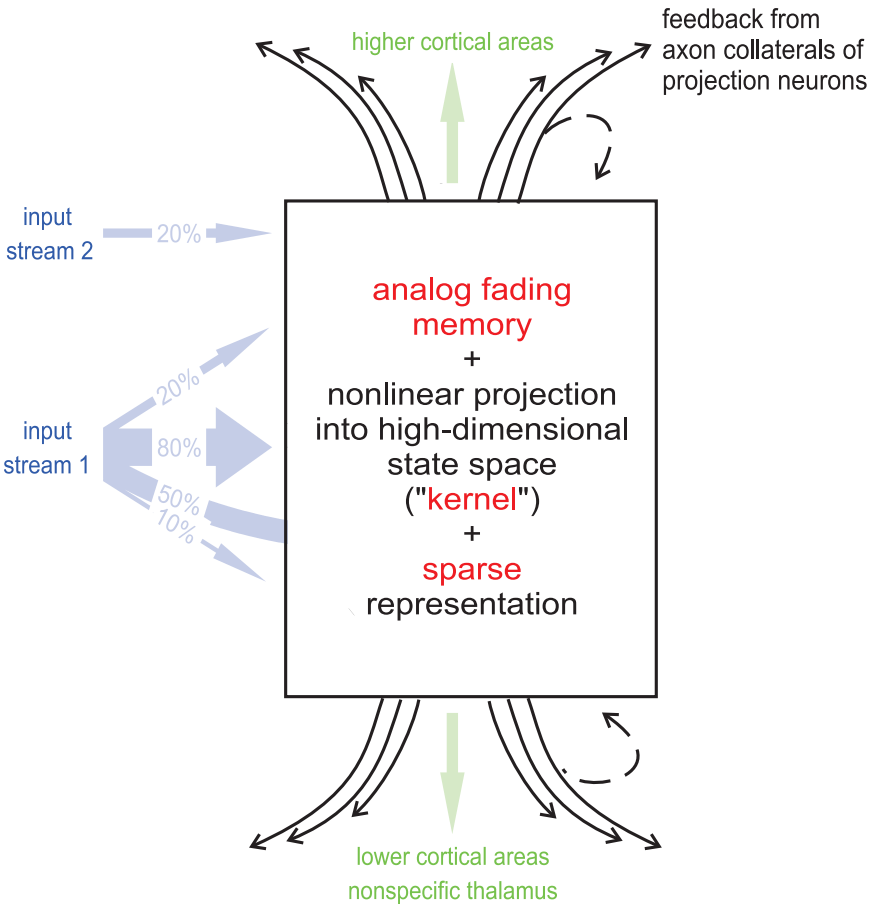

Fig. 1.2.: Hypothetical computational function of a generic cortical microcircuit in the context of the LSM model. In general the projection neurons also provide feedback back into the microcircuit (see Theorem 1.2 in section 3).

cessing for a large range of possible tasks of different readouts. It could also be adaptive, but by other learning algorithms than the readouts, for example by unsupervised learning algorithms that are directed by the statistics of the inputs $u(t)$ to the Liquid. The Liquid is in more abstract models a generic dynamical system – preferentially consisting of diverse rather than uniform and stereotypical components (for reasons that will become apparent below). In biological models (see Fig. 1.1, Fig. 1.2, Fig. 1.3) the Liquid is typically a generic recurrently connected local network of neurons, modelling for example a cortical column which spans all cortical layers and

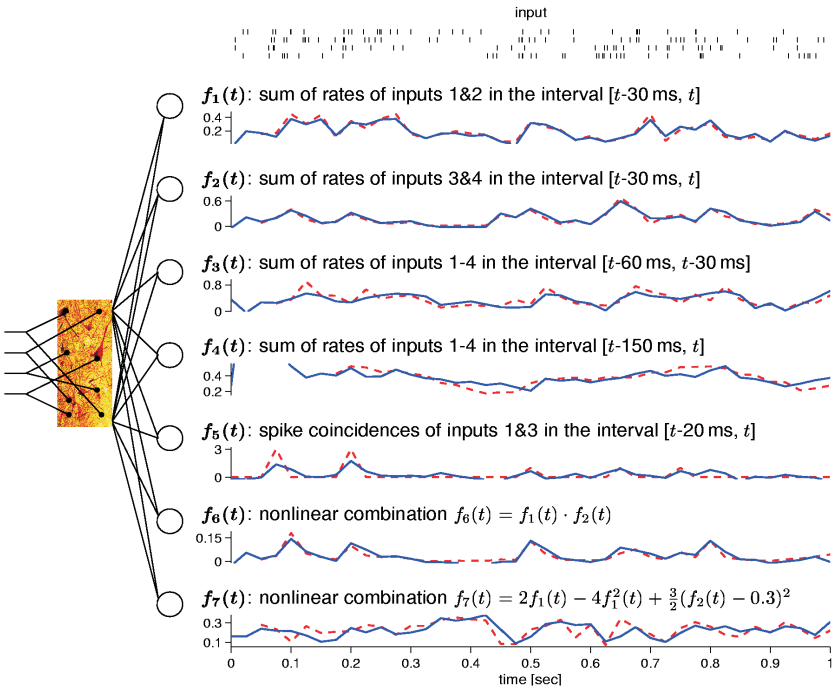

Fig. 1.3.: Multi-tasking in real-time. Below the 4 input spike trains (shown at the top) the target outputs (dashed curves) and actual outputs (solid curves) of 7 linear readout neurons are shown in real-time (on the same time axis). Targets were to output every 30 ms the sum of the current firing rates of input spike trains 1 and 2 during the preceding 30 ms (f_1), the sum of the current firing rates of input spike trains 3 and 4 during the preceding 30 ms (f_2), the sum of f_1 and f_2 in an earlier time interval [t-60 ms, t-30 ms] (f_3) and during the interval [t-150 ms, t] (f_4), spike coincidences between inputs 1&3 ($f_5(t)$ is defined as the number of spikes which are accompanied by a spike in the other spike train within 5 ms during the interval [t-20 ms, t]), a simple nonlinear combination f_6 (product) and a randomly chosen complex nonlinear combination f_7 of earlier described values. Since all readouts were linear units, these nonlinear combinations are computed implicitly within the generic microcircuit model (consisting of 270 spiking neurons with randomly chosen synaptic connections). The performance of the model is shown for test spike inputs that had not been used for training (see [5] for details).

has a diameter of about 0.5 mm. But it has been shown that also an actual physical Liquid (such as a bucket of water) may provide an important computational preprocessing for subsequent linear readouts (see [6] for a demonstration, and [7] for theoretical analysis). We refer to the input vec-

tor $\mathbf{x}(t)$ that a readout receives from a Liquid at a particular time point t as the liquid state (of the Liquid) at this time point t (in terms of dynamical systems theory, this liquid state is that component of the internal state of the Liquid - viewed as a dynamical system - that is visible to some readout unit). This notion is motivated by the observation that the LSM generalizes the information processing capabilities of a finite state machine (which also maps input functions onto output functions, although these are functions of discrete time) from a finite to a continuous set of possible values, and from discrete to continuous time. Hence the states $\mathbf{x}(t)$ of an LSM are more "liquid" than those of a finite state machine.

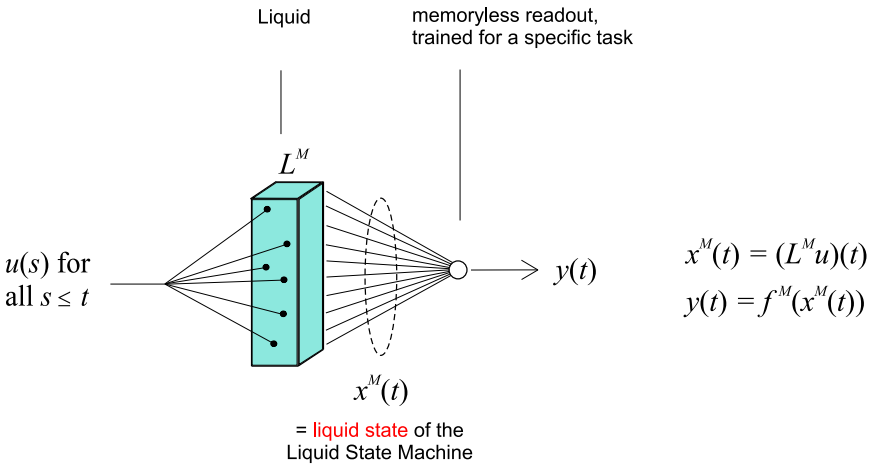

Fig. 1.4.: Structure of a Liquid State Machine (LSM) M, which transforms input streams $u(\cdot)$ into output streams $y(\cdot)$. L^M denotes a Liquid (e.g., some dynamical system), and the "liquid state" $\mathbf{x}^M(t) \in \mathbb{R}^k$ is the input to the readout at time t. More generally, $\mathbf{x}^M(t)$ is that part of the current internal state of the Liquid that is "visible" for the readout. Only one input- and output channel are shown for simplicity.

This architecture of a LSM, consisting of Liquid and readouts, makes sense, because it turns out that in many contexts there exist common computational preprocessing needs for many different readouts with different computational goals. This can already be seen from the trivial fact that computing all pairwise products of a set of input numbers (say: of all components of a multi-dimensional input $u(t')$ for a fixed time point t') gives any subsequent linear readout the virtual expressive power of any quadratic computation on the original input $u(t')$. A pre-processor for a linear read-

out is even more useful if it maps more generally any frequently occurring (or salient) different input streams $u(\cdot)$ onto linearly independent liquid states $\mathbf{x}(t)$ [8], similarly as an RBF-kernel for Support Vector Machines. A remarkable aspect of this more general characterization of the preprocessing task for a Liquid is that it does not require that it computes precise products, or any other concrete nonlinear mathematical operation. Any "found" analog computing device (it could even be very imprecise, with mismatched transistors or other more easily found nonlinear operations in physical objects) consisting of sufficiently diverse local processes, tends to approximate this requirement quite well. A closer look shows that the actual requirement on a Liquid is a bit more subtle, since one typically only wants that the Liquid maps "saliently" different input streams $u(\cdot)$ onto linearly independent liquid states $\mathbf{x}(t)$, whereas noisy variations of the "same" input stream should rather be mapped onto a lower dimensional manifold of liquid states, see [8, 9] for details.

An at least equally important computational preprocessing task of a Liquid is to provide all temporal integration of information that is needed by the readouts. If the target value $y(t)$ of a readout at time t depends not only on the values of the input streams at the same time point t, but on a range of input values $u(s)$ for many different time points s (say, if $y(t)$ is the integral over one component of $u(s)$ for a certain interval $[t-1,\ t]$), then the Liquid has to collect all required information from inputs at preceding time points $u(s)$, and present all this information simultaneously in the liquid state $\mathbf{x}(t)$ at time point t (see Fig. 1.3 and Fig. 1.4). This is necessary, because the readout stage has by assumption no temporal integration capability of its own, i.e., it can only learn to carry out "static" computations that map $\mathbf{x}(t)$ onto $y(t)$. A readout does not even know what the current time t is. It just learns a map f from input numbers to output numbers. Hence it just learns a fixed recoding (or projection) f from liquid states into output values. This severe computational limitation of the readout of a LSM is motivated by the fact, that learning a static map f is so much simpler than learning a map from input streams to output streams. And a primary goal of the LSM is to make the learning as fast and robust as possible. Altogether, an essential prediction of LSM-theory for information processing in cortical microcircuits is that they accumulate information over time. This prediction has recently been verified for cortical microcircuits in primary visual cortex [11] and in the primary auditory cortex [12].

The advantage of choosing for a LSM the simplest possible learning device is twofold: Firstly, learning for a single readout neuron is fast, and

cannot get stuck in local minima (like backprop or EM). Secondly, the simplicity of this learning device entails a superior – in fact, arguably optimal – generalization capability of learned computational operations to new inputs streams. This is due to the fact that its VC-dimension (see [10] for a review) is equal to the dimensionality of its input plus 1. This is the smallest possible value of any nontrivial learning device with the same input dimension.

It is a priori not clear that a Liquid can carry the highly nontrivial computational burden of not only providing all desired nonlinear preprocessing for linear readouts, but simultaneously also all temporal integration that they might need in order to implement a particular mapping from input streams $u(\cdot)$ onto output streams $y(\cdot)$. But there exist two basic mathematical results (see Theorems 1.1 and 1.2 in Sec. 1.3) which show that this goal can in principle be achieved, or rather approximated, by a concrete physical implementation of a Liquid which satisfies some rather general property. A remarkable discovery, which had been achieved independently and virtually simultaneously around 2001 by Herbert Jaeger [13], is that there are surprisingly simple Liquids, i.e., generic preprocessors for a subsequent linear learning device, that work well independently of the concrete computational tasks that are subsequentially learned by the learning device. In fact, naturally found materials and randomly connected circuits tend to perform well as Liquids, which partially motivates the interest of the LSM model both in the context of computations in the brain, and in novel computing technologies.

Herbert Jaeger [13] had introduced the name Echo State Networks (ESNs) for the largely equivalent version of the LSM that he had independently discovered. He explored applications of randomly connected recurrent networks of sigmoidal neurons without noise as Liquids (in contrast to the biologically oriented LSM studies, that assume significant internal noise in the Liquid) to complex time series prediction tasks, and showed that they provide superior performance on common benchmark tasks. The group of Benjamin Schrauwen (see [14–17]) introduced the term Reservoir Computing as a more general term for the investigation of LSMs, ESNs and variations of these models. A variety of applications of these models can be found in a special issue of Neural Networks 2007 (see [18]). All these groups are currently collaborating in the integrated EU-project ORGANIC (= Self-organized recurrent neural learning for language processing) that investigates applications of these models to speech understanding and reading of handwritten text (see

`http://reservoir-computing.org`). An industrial partner in this project, the company PLANET (`http://english.planet.de`) had already good success in applications of Reservoir Computing to automated high-speed reading of hand-written postal addresses.

We will contrast these models and their computational use with that of Turing machines in the next section. In Sec. 1.3 we will give a formal definition of the LSM, and also some theoretical results on its computational power. We will discuss applications of the LSM and ESN model to biology and new computing devices in Sec. 1.4 (although the discussion of its biological aspects will be very short in view of the recent review paper [19] on this topic).

1.2. Why Turing machines are not useful for many important computational tasks

The computation of a Turing machine always begins in a designated initial state q_0, with the input x (some finite string of symbols from some finite alphabet) written on some designated tape. The computation runs until a halt-state is entered (the inscription y of some designated tape segment is then interpreted as the result of the computation). This is a typical example for an *offline computation* (Fig. 1.5A), where the complete input x is available at the beginning of the computation, and no trace of this computation, or of its result y, is left when the same Turing machine subsequently carries out another computation for another input $\tilde{x}$ (starting again in state q_0). In contrast, the result of a typical computation in the neuronal system of a biological organism, say the decision about the location y on the ground where the left foot is going to be placed at the next step (while walking or running), depends on several pieces of information: on information from the visual system, from the vestibular system which supports balance control, from the muscles (proprioceptive feedback about their current state), from short term memory (how well did the previous foot placement work?), from long term memory (how slippery is this path at the current weather condition?), from brain systems that have previously decided where to go and at what speed, and on information from various other parts of the neural system. In general these diverse pieces of information arrive at different points in time, and the computation of y has to start before the last one has come in (see Fig. 1.5B). Furthermore, new information (e.g., visual information and proprioceptive feedback) arrives continuously, and it is left up to the computational system how much of it can be integrated

into the computation of the position y of the next placement of the left foot (obviously those organisms have a better chance to survive which also can integrate later arriving information into the computation). Once the computation of y is completed, the computation of the location y' where the right foot is subsequently placed is not a separate computation, that starts again in some neutral initial state q_0. Rather, it is likely to build on pieces of inputs and results of subcomputations that had already been used for the preceding computation of y.

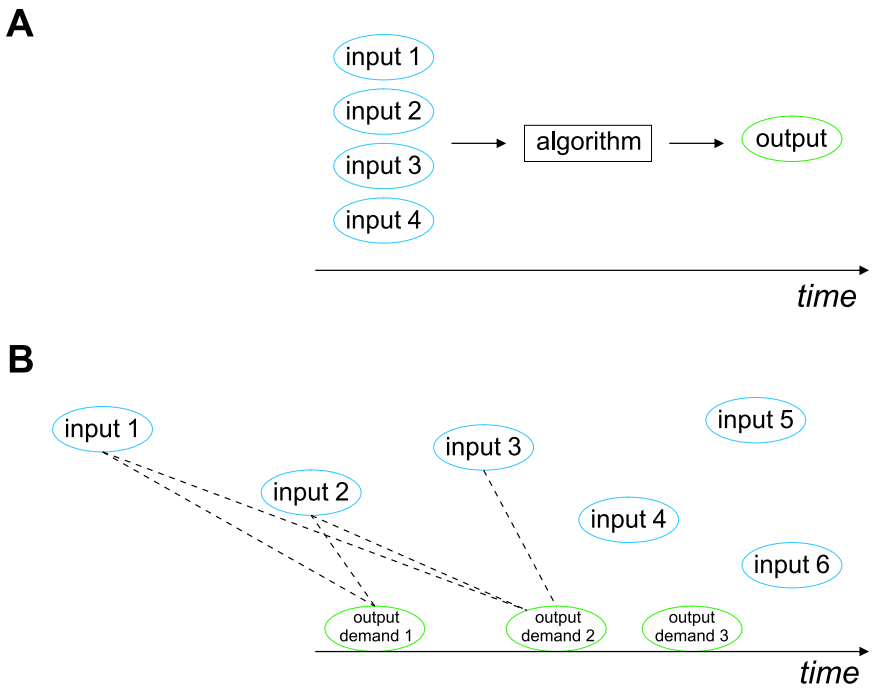

Fig. 1.5.: Symbolic representation of offline and online computations. **(A)** In an offline computation all relevant input computations are available at the start of the computation, and the algorithm may require substantial computation time until the result becomes available. **(B)** In online computations additional pieces of information arrive all the time. The most efficient computational processing scheme integrates as many preceding input pieces as possible into its output whenever an output demand arises. In that sense computations by a LSM are optimally efficient.

The previously sketched computational task is a typical example for an *online computation* (where input pieces arrive all the time, not in one

batch), see Fig. 1.5B. Furthermore it is an example for a *real-time computation*, where one has a strict deadline by which the computation of the output y has to be completed (otherwise a 2-legged animal would fall). In fact, in some critical situations (e.g., when a 2-legged animal stumbles, or hits an unexpected obstacle) a biological organism is forced to apply an *anytime algorithm*, which tries to make optimal use of intermediate results of computational processing that has occurred up to some externally given time point t_0 (such forced halt of the computation could occur at "any time"). Difficulties in the control of walking for 2-legged robots have taught us how difficult it is to design algorithms which can carry out this seemingly simple computational task. In fact, this computational problem is largely unsolved, and humanoid robots can only operate within environments for which they have been provided with an accurate model. This is perhaps surprising, since on the other hand current computers can beat human champions in seemingly more demanding computational tasks, such as winning a game of chess. One might argue that one reason, why walking in a new terrain is currently a computationally less solved task, is that computation theory and algorithm design have focused for several decades on offline computations, and have neglected seemingly mundane computational tasks such as walking. This bias is understandable, because evolution had much more time to develop a computational machinery for the control of human walking, and this computational machinery works so well that we don't even notice anymore how difficult this computational task is.

1.3. Formal Definition and Theory of Liquid State Machines

A computation machine M that carries out online computations typically computes a function F that does not map input numbers or (finite) bit strings onto output numbers or bit strings, but input streams onto output streams. These input- and output streams are usually encoded as functions $u : \mathbb{Z} \to \mathbb{R}^n$ or $u : \mathbb{R} \to \mathbb{R}^n$, where the argument t of $u(t)$ is interpreted as the (discrete or continuous) time point t when the information that is encoded by $u(t) \in \mathbb{R}^n$ becomes available. Hence such computational machine M computes a function of higher type (usually referred to as operator, functional, or filter), that maps input functions u from some domain U onto output functions y. In lack of a better term we will use the term filter in this section, although filters are often associated with somewhat trivial signal processing or preprossessing devices. However, one should not fall into the trap of identifying the general term of a filter with special classes

of filters such as linear filters. Rather one should keep in mind that any input to any organism is a function of time, and any motor output of an organism is a function of time. Hence biological organisms compute filters. The same holds true for any artificial behaving system, such as a robot.

We will only consider computational operations on functions of time that are input-driven, in the sense that the output does not depend on any absolute internal clock of the computational device. Filters that have this property are called time invariant. Formally one says that a filter F is *time invariant* if any temporal shift of the input function $u(\cdot)$ by some amount t_0 causes a temporal shift of the output function by the same amount t_0, i.e., $(Fu^{t_0})(t) = (Fu)(t+t_0)$ for all $t, t_0 \in \mathbb{R}$, where u^{t_0} is the function defined by $u^{t_0}(t) := u(t+t_0)$. Note that if the domain U of input functions $u(\cdot)$ is closed under temporal shifts, then a time invariant filter $F : U \to \mathbb{R}^{\mathbb{R}}$ is identified uniquely by the values $y(0) = (Fu)(0)$ of its output functions $y(\cdot)$ at time 0. In other words: in order to identify or characterize a time invariant filter F we just have to observe its output values at time 0, while its input varies over all functions $u(\cdot) \in U$. Hence one can replace in the mathematical analysis such filter F by a functional, i.e., a simpler mathematical object that maps input functions onto real values (rather than onto functions of time).

Various theoretical models for analog computing are of little practical use because they rely on hair-trigger decisions, for example they allow that the output is 1 if the value of some real-valued input variable u is ≥ 0, and 0 otherwise. Another unrealistic aspect of some models for computation on functions of time is that they automatically allow that the output of the computation depends on the full infinitely long history of the input function $u(\cdot)$. Most practically relevant models for analog computation on continuous input streams degrade gracefully under the influence of noise, i.e., they have a fading memory. *Fading memory* is a continuity property of filters F, which requires that for any input function $u(\cdot) \in U$ the output $(Fu)(0)$ can be approximated by the outputs $(Fv)(0)$ for any other input functions $v(\cdot) \in U$ that approximate $u(\cdot)$ on a sufficiently long time interval $[-T, 0]$ in the past. Formally one defines that $F : U \to \mathbb{R}^{\mathbb{R}}$ has fading memory if for every $u \in U^n$ and every $\varepsilon > 0$ there exist $\delta > 0$ and $T > 0$ so that $|(Fv)(0) - (Fu)(0)| < \varepsilon$ for all $v \in U$ with $\|u(t) - v(t)\| < \delta$ for all $t \in [-T, 0]$. Informally, a filter F has fading memory if the most significant bits of its current output value $(Fu)(0)$ depend just on the most significant bits of the values of its input function $u(\cdot)$ in some finite time interval $[-T, 0]$. Thus, in order to compute the most significant bits of $(Fu)(0)$ it is

not necessary to know the *precise* value of the input function $u(s)$ for any time s, and it is also not necessary to have knowledge about values of $u(\cdot)$ for more than a finite time interval back into the past.

The universe of time-invariant fading memory filters is quite large. It contains all filters F that can be characterized by Volterra series, i.e., all filters F whose output $(Fu)(t)$ is given by a finite or infinite sum (with $d = 0, 1, \ldots$) of terms of the form $\int_0^\infty \ldots \int_0^\infty h_d(\tau_1, \ldots, \tau_d) \cdot u(t - \tau_1) \cdot \ldots \cdot u(t - \tau_d) d\tau_1 \ldots d\tau_d$, where some integral kernel h_d is applied to products of degree d of the input stream $u(\cdot)$ at various time points $t - \tau_i$ back in the past. In fact, under some mild conditions on the domain U of input streams the class of time invariant fading memory filters coincides with the class of filters that can be characterized by Volterra series.

In spite of their complexity, all these filters can be uniformly approximated by the simple computational models M of the type shown in Fig. 1.4, which had been introduced in [1]:

Theorem 1.1. (based on [20]; see Theorem 3.1 in [21] for a detailed proof). *Any filter F defined by a Volterra series can be approximated with any desired degree of precision by the simple computational model M shown in Fig. 1.1 and Fig. 1.2.*

- *if there is a rich enough pool* $\mathbf{B}$ *of basis filters (time invariant, with fading memory) from which the basis filters $B_1, \ldots, B_k$ in the filterbank L^M can be chosen (*$\mathbf{B}$ *needs to have the pointwise separation property) and*
- *if there is a rich enough pool* $\mathbf{R}$ *from which the readout functions f can be chosen (*$\mathbf{R}$ *needs to have the universal approximation property, i.e., any continuous function on a compact domain can be uniformly approximated by functions from* $\mathbf{R}$*).*

Definition 1.1. *A* class $\mathbf{B}$ of basis filters has the pointwise separation property if there exists for any two input functions $u(\cdot), v(\cdot)$ with $u(s) \neq v(s)$ for some $s \leq t$ a basis filter $B \in \mathbf{B}$ with $(Bu)(t) \neq (Bv)(t)$.

It turns out that many real-world dynamical systems (even a pool of water) satisfy (for some domain U of input streams) at least some weak version of the pointwise separation property, where the outputs $\mathbf{x}^M(t)$ of the basis filters are replaced by some "visible" components of the state vector of the dynamical system. In fact, many real-world dynamical systems also satisfy approximately an interesting kernel property[a], which makes

[a] A kernel (in the sense of machine learning) is a nonlinear projection Q of n input

it practically sufficient to use just a *linear* readout function f^M. This is particularly important if L^M is kept fixed, and only the readout f^M is selected (or trained) in order to approximate some particular Volterra series F. Reducing the adaptive part of M to the *linear* readout function f^M has the unique advantage that a learning algorithm that uses gradient descent to minimize the approximation error of M cannot get stuck in local minima of the mean-squared error. The resulting computational model can be viewed as a generalization of a finite state machine to continuous time and continuous ("liquid") internal states $\mathbf{x}^M(t)$. Hence it is called a Liquid State Machine.

If the dynamical systems L^M have fading memory, then only filters with fading memory can be represented by the resulting LSMs. Hence they cannot approximate arbitrary finite state machines (not even for the case of discrete time and a finite range of values $u(t)$). It turns out that a large jump in computational power occurs if one augments the computational model from Fig. 1.4 by a feedback from a readout back into the circuit (assume it enters the circuit like an input variable).

Theorem 1.2. [22]. *There exists a large class S_n of dynamical systems C with fading memory (described by systems of n first order differential equations) that acquire through feedback universal computational capabilities for analog computing. More precisely: through a proper choice of a (memoryless) feedback function K and readout h they can simulate any given dynamical system of the form $z^{(n)} = G(z, z', \ldots, z^{(n-1)}) + u$ with a sufficiently smooth function G (see Fig. 1.6). This holds in particular for neural circuits C defined by differential equations of the form $x_i'(t) = -\lambda_i x_i(t) + \sigma(\sum_{j=1}^n a_{ij} x_j(t)) + b_i \cdot \sigma(v(t))$ (under some conditions on the λ_i, a_{ij}, b_i).*

If one allows several feedbacks K, such dynamical systems C become universal for n^{th} order dynamical systems defined by a system consisting of a corresponding number of differential equations. Since such systems of differential equations can simulate arbitrary Turing machines [23], these dynamical systems C with a finite number of feedbacks become (according

variables $u_1, \ldots, u_n$ into some high-dimensional space. For example all products $u_i \cdot u_j$ could be added as further components to the n-dimensional input vector $< u_1, \ldots, u_n >$. Such nonlinear projection Q boosts the power of any *linear* readout f applied to $Q(\mathbf{u})$. For example in the case where $Q(\mathbf{u})$ contains all products $u_i \cdot u_j$, a subsequent linear readout has the same expressive capability as quadratic readouts f applied to the original input variables $u_1, \ldots, u_n$. More abstractly, Q should map all inputs $\mathbf{u}$ that need to be separated by a readout onto a set of linearly independent vectors $Q(\mathbf{u})$.

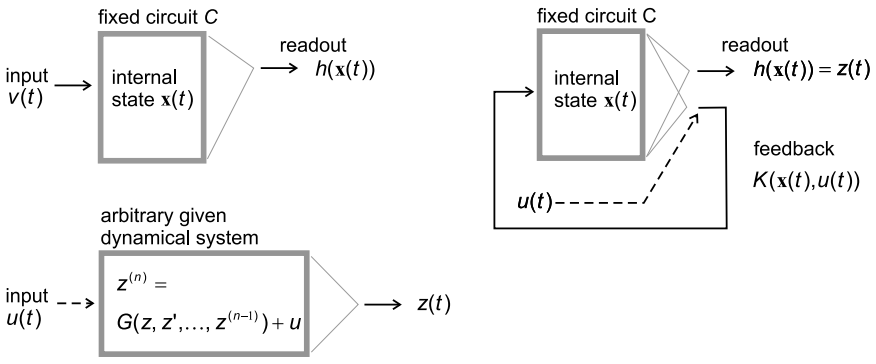

Fig. 1.6.: Illustration of the notation and result of Theorem 1.2

.

to the Church-Turing thesis) also universal for *digital computation*.

Theorem 1.2 suggests that even quite simple neural circuits with feedback have in principle unlimited computational power[b]. This suggests that the main problem of a biological organism becomes the *selection* (or learning) of suitable feedback functions K and readout functions h. For dynamical systems C that have a good kernel-property, already *linear* feedbacks and readouts endow such dynamical systems with the capability to emulate a fairly large range of other dynamical systems (or "analog computers").

Recent theoretical work has addressed methods for replacing supervised training of readouts by reinforcement learning [24] (where readout neurons explore autonomously different settings of their weights, until they find some which yield outputs that are rewarded) and by completely unsupervised learning (where not even rewards for successful outputs are available). It is shown in [25] that already the repeated occurrence of certain trajectories of liquid status enables a readout to classify such trajectories according to the type of input which caused them. In this way a readout can for example learn without supervision to classify (i.e., "understand") spoken digits. The theoretical basis for this result is that unsupervised slow feature extraction approximates the discrimination capability of the Fisher Linear Discriminant if the sequence of liquid states that occur during training satisfies a certain statistical condition.

[b]Of course, in the presence of noise this computational power is reduced to that of a finite state machine, see [22] for details.

1.4. Applications

LSMs had been introduced in the process of searching for computational models that can help us to understand the computations that are carried out in a "cortical microcircuit" [26], i.e., in a local circuit of neurons in the neocortex (say in a "cortical column"). This approach has turned out to be quite successful, since it made it possible to carry out quite demanding computations with circuits consisting of reasonably realistic models for biological neurons ("spiking neurons") and biological synapses ("dynamical synapses"). Note that in this model a large number of different readout neurons can learn to extract different information from the same circuit. One concrete benchmark task that has been considered was the classification ("recognition") of spoken digits [27]. It turned out that already a LSM where the "Liquid" consisted of a randomly connected circuit of just 135 spiking neurons performed quite well. In fact, it provided a nice example for "anytime computations", since the linear readout could be trained effectively to guess at "any time", while a digit was spoken, the proper classification of the digit [1, 5]. More recently it has been shown that with a suitable transformation of spoken digits into spike trains one can achieve with this simple method the performance level of state-of-the-art algorithms for speech recognition [14].

A number of recent neurobiological experiments in vivo has lead many biologists to the conclusion, that also for neural computation in larger neural systems than cortical microcircuits a new computational model is needed (see the recent review [28]). In this new model certain frequently occurring trajectories of network states – rather than attractors to which they might or might not converge – should become the main carriers of information about external sensory stimuli. The review [19] examines to what extent the LSM and related models satisfy the need for such new models for neural computation.

It has also been suggested [29] that LSMs might present a useful framework for modeling computations in gene regulation networks. These networks also compute on time varying inputs (e.g., external signals) and produce a multitude of time varying output signals (transcription rates of genes). Furthermore these networks are composed of a very large number of diverse subprocesses (transcription of transcription factors) that tend to have each a somewhat different temporal dynamics (see [30]). Hence they exhibit characteristic features of a Liquid in the LSM model. Furthermore there exist perceptron-like gene regulation processes that could serve as

readouts from such Liquids (see chapter 6 in [30]).

In the remainder of this section I will review a few applications of the LSM model to the design of new artificial computing system. In [6] it had been demonstrated that one can use a bucket of water as Liquid for a physical implementation of the LSM model. Input streams were injected via 8 motors into this Liquid and video-images of the surface of the water were used as "liquid states" $\mathbf{x}(t)$. It was demonstrated in [6] that the previously mentioned classification task of spoken digits could in principle also be carried out with this – certainly very innovative – computing device. But other potential technological applications of the LSM model have also been considered. The article [15] describes an implementation of a LSM in FPGAs (Field Programmable Gate Arrays). In the USA a patent was recently granted for a potential implementation of a LSM via nanoscale molecular connections [31]. Furthermore work is in progress on implementations of LSMs in photonic computing, where networks of semiconductor optical amplifiers serve as Liquid (see [16] for a review).

The exploration of potential engineering applications of the computational paradigm discussed in this article is simultaneously also carried out for the closely related echo state networks (ESNs) [13], where one uses simpler non-spiking models for neurons in the "Liquid", and works with high numerical precision in the simulation of the "Liquid" and the training of linear readouts. Research in recent years has produced quite encouraging results regarding applications of ESNs and LSMs to problems in telecommunication [13], robotics [32], reinforcement learning [33], natural language understanding [34], as well as music-production and -perception [35].

1.5. Discussion

We have argued in this article that Turing machines are not well-suited for modeling computations in biological neural circuits, and proposed Liquid state machines (LSMs) as a more adequate modeling framework. They are designed to model real-time computations (as well as anytime computations) on continuous input streams. In fact, it is quite realistic that a LSM can be trained to carry out the online computation task that we had discussed in Sec. 1.2 (see [36] for a first application to motor control). A characteristic feature of practical implementations of the LSM model is that its "program" consists of the weights $\mathbf{w}$ of a linear readout function. These weights provide suitable targets for learning (while all other parameters of the LSM can be fixed in advance, based on the expected complexity and

precision requirement of the computational tasks that are to be learnt). It makes a lot of sense (from the perspective of statistical learning theory) to restrict learning to such weights $\mathbf{w}$, since they have the unique advantage that gradient descent with regard to some mean-square error function $E(\mathbf{w})$ cannot get stuck in local minima of this error function (since $\nabla_{\mathbf{w}} E(\mathbf{w}) = \mathbf{0}$ defines an affine – hence connected – subspace of the weight space for a linear learning device).

One can view these weights $\mathbf{w}$ of the linear readout of a LSM as an analogon to the code $< M >$ of a Turing machine M that is simulated by a universal Turing machine. This analogy makes the learning advantage of LSMs clear, since there is no efficient learning algorithm known which allows us to learn the program $< M >$ for a Turing machine M from examples for correct input/output pairs of M. However the examples discussed in this chapter show that an LSM can be trained quite efficiently to approximate a particular map from input to output streams.

We have also shown in Theorem 1.2 that LSMs can overcome the limitation of a fading memory if one allows feedback from readouts back into the "Liquid". Then not only all digital, but (in a well-defined sense) also all analog computers can be simulated by a fixed LSM, provided that one is allowed to vary the readout functions (including those that provide feedback). Hence these readout functions can be viewed as program for the simulated analog computers (note that all "readout functions" are just "static" functions, i.e., maps from $\mathbb{R}^n$ into $\mathbb{R}$, whereas the LSM itself maps input streams onto output streams). In those practically relevant cases that have been considered so far, these readout functions could often be chosen to be linear. A satisfactory characterization of the computational power that can be reached with linear readouts is still missing. But obviously the kernel-property of the underlying "Liquid" can boost the richness of the class of analog computers that can be simulated by a fixed LSM with linear readouts.

The theoretical analysis of computational properties of randomly connected circuits and other potential "Liquids" is still in its infancy. We refer to [7–9, 17, 37] for useful first steps. The qualities that we expect from the "Liquid" of a LSM are completely different from those that one expects from standard computing devices. One expects diversity (rather than uniformity) of the responses of individual gates within a Liquid (see Theorem 1.1), as well as diverse local dynamics instead of synchronized local gate operations. Achieving such diversity is apparently easy to attain by biological neural circuits and by new artificial circuits on the molecular

or atomic scale. It is obviously much easier to attain than an emulation of precisely engineered and synchronized circuits of the type that we find in our current generation of digital computers. These only function properly if all local units are identical copies of a small number of template units that respond in a stereotypical fashion. For a theoretician it is also interesting to learn that sparse random connections within a recurrent circuit turn out to provide better computational capabilities to a LSM than those connectivity graphs that have primarily been considered in earlier theoretical studies, such as all-to-all connections (Hopfield networks) or a 2-dimensional grid (which is commonly used for cellular automata). Altogether one sees that the LSM and related models provide a wide range of interesting new problems in computational theory, the chance to understand biological computations, and new ideas for the invention of radically different artificial computing devices that exploit, rather than suppress, inherent properties of diverse physical substances.

Acknowledgements

Partially supported by the Austrian Science Fund FWF, project P17229 and project S9102-N13, and by the European Union, project # FP6-015879 (FACETS), project FP7-216593 (SECO) and FP7-231267 (ORGANIC).

References

[1] W. Maass, T. Natschlaeger, and H. Markram, Real-time computing without stable states: A new framework for neural computation based on perturbations, *Neural Computation*. **14**(11), 2531–2560, (2002).

[2] A. M. Thomson, D. C. West, Y. Wang, and A. P. Bannister, Synaptic connections and small circuits involving excitatory and inhibitory neurons in layers 2 - 5 of adult rat and cat neocortex: triple intracellular recordings and biocytin labelling in vitro, *Cerebral Cortex*. **12**(9), 936–953, (2002).

[3] S. Haeusler and W. Maass, A statistical analysis of information processing properties of lamina-specific cortical microcircuit models, *Cerebral Cortex*. **17**(1), 149–162, (2007).

[4] S. Haeusler, K. Schuch, and W. Maass, Motif distribution and computational performance of two data-based cortical microcircuit templates, *J. of Physiology (Paris)*. (2009). in press.

[5] W. Maass, T. Natschlaeger, and H. Markram, Fading memory and kernel properties of generic cortical microcircuit models, *Journal of Physiology – Paris*. **98**(4–6), 315–330, (2004).

[6] C. Fernando and S. Sojakka. Pattern recognition in a bucket: a real liquid brain. In *Proceedings of ECAL*. Springer, (2003).

[7] S. Ganguli, D. Huh, and H. Sompolinsky, Memory traces in dynamical systems, *PNAS USA*. **105**, 18970–18975, (2008).
[8] R. Legenstein and W. Maass, Edge of chaos and prediction of computational performance for neural microcircuit models, *Neural Networks*. **20**(3), 323–334, (2007).
[9] R. Legenstein and W. Maass. What makes a dynamical system computationally powerful? In eds. S. Haykin, J. C. Principe, T. Sejnowski, and J. McWhirter, *New Directions in Statistical Signal Processing: From Systems to Brains*, pp. 127–154. MIT Press, (2007).
[10] P. L. Bartlett and W. Maass. Vapnik-Chervonenkis dimension of neural nets. In ed. M. A. Arbib, *The Handbook of Brain Theory and Neural Networks*, pp. 1188–1192. MIT Press (Cambridge), 2nd edition, (2003).
[11] D. Nikolic, S. Haeusler, W. Singer, and W. Maass, Distributed fading memory for stimulus properties in the primary visual cortex, *PLoS Biology*. **7**(12), 1–19, (2009).
[12] S. Klampfl, S.V. David, P. Yin, S.A. Shamma, and W. Maass, Integration of stimulus history in information conveyed by neurons in primary auditory cortex in response to tone sequences, *39th Annual Conference of the Society for Neuroscience, Program 163.8, Poster T6*. (2009).
[13] H. Jaeger and H. Haas, Harnessing nonlinearity: predicting chaotic systems and saving energy in wireless communication, *Science*. **304**, 78–80, (2004).
[14] D. Verstraeten, B. Schrauwen, D. Stroobandt, and J. V. Campenhout, Isolated word recognition with the liquid state machine: a case study., *Information Processing Letters*. **95**(6), 521–528, (2005).
[15] B. Schrauwen, M. D'Haene, D. Verstraeten, and D.Stroobandt, Compact hardware liquid state machines on FPGA for real-time speech recognition, *Neural Networks*. **21**, 511–523, (2008).
[16] K. Vandoorne, W. Dierckx, B. Schrauwen, D. Verstraeten, R. Baets, P. Bienstman, and J. V. Campenhout, Toward optical signal processing using photonic reservoir computing, *Optics Express*. **16**(15), 11182–11192, (2008).
[17] B. Schrauwen, L. Buesing, and R. Legenstein. On computational power and the order-chaos phase transition in reservoir computing. In *Proceeding of NIPS 2008, Advances in Neural Information Processing Systems*. MIT Press, (2009). in press.
[18] H. Jaeger, W. Maass, and J. Principe, Introduction to the special issue on echo state networks and liquid state machines, *Neural Networks*. **20**(3), 287–289, (2007).
[19] D. Buonomano and W. Maass, State-dependent computations: Spatiotemporal processing in cortical networks., *Nature Reviews in Neuroscience*. **10** (2), 113–125, (2009).
[20] S. Boyd and L. O. Chua, Fading memory and the problem of approximating nonlinear oparators with Volterra series, *IEEE Trans. on Circuits and Systems*. **32**, 1150–1161, (1985).
[21] W. Maass and H. Markram, On the computational power of recurrent circuits of spiking neurons, *Journal of Computer and System Sciences*. **69**(4), 593–616, (2004).

[22] W. Maass, P. Joshi, and E. D. Sontag, Computational aspects of feedback in neural circuits, *PLoS Computational Biology*. **3**(1), e165, 1–20, (2007).
[23] M. S. Branicky, Universal computation and other capabilities of hybrid and continuous dynamical systems, *Theoretical Computer Science*. **138**, 67–100, (1995).
[24] 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), 1–27, (2008).
[25] S. Klampfl and W. Maass, A neuron can learn anytime classification of trajectories of network states without supervision, *submitted for publication*. (Feb. 2009).
[26] W. Maass and H. Markram. Theory of the computational function of microcircuit dynamics. In eds. S. Grillner and A. M. Graybiel, *The Interface between Neurons and Global Brain Function*, Dahlem Workshop Report 93, pp. 371–390. MIT Press, (2006).
[27] J. J. Hopfield and C. D. Brody, What is a moment? Transient synchrony as a collective mechanism for spatio-temporal integration, *Proc. Nat. Acad. Sci. USA*. **98**(3), 1282–1287, (2001).
[28] M. Rabinovich, R. Huerta, and G. Laurent, Transient dynamics for neural processing, *Science*. **321**, 45–50, (2008).
[29] B. Jones, D. Stekel, J. Rowe, and C. Fernando, Is there a liquid state machine in the bacterium escherichia coli?, *Artificial Life*. ALIFE'07, IEEE Symposium, 187–191, (2007).
[30] U. Alon, *An Introduction to Systems Biology: Design Principles of Biological Circuits*. (Chapman & Hall, 2007).
[31] A. Nugent, *Physical neural network liquid state machine utilizing nanotechnology*. (US-Patent 7 392 230 32, June 2008).
[32] J. Hertzberg, H. Jäger, and F. Schönherr. Learning to ground fact symbols in behavior-based robot. In ed. F. van Harmelen ed., *Proc. of the 15th European Conference on Artificial Intelligence.*, pp. 708–712, Amsterdam, (2002). IOS Press.
[33] K. Bush and C. Anderson. Modeling reward functions for incomplete state representations via echo state networks. In *Proceedings of the International Joint Conference on Neural Networks, Montreal, Quebec*, (2005).
[34] M. Tong, A. Bickett, E. Christiansen, and G. Cotrell, Learning grammatical structure with echo state networks, *Neural Networks*. **20**(3), 424–432, (2007).
[35] H. Jaeger and D. Eck. Can't get you out of my head: A connectionist model of cyclic rehearsal. In eds. I. Wachsmuth and G. Knoblich, *Modeling Communication with Robots and Virtual Humans*, (2008).
[36] P. Joshi and W. Maass, Movement generation with circuits of spiking neurons, *Neural Computation*. **17**(8), 1715–1738, (2005).
[37] O. L. White, D. D. Lee, and H. Sompolinsky, Short-term memory in orthogonal neural networks, *Phys. Rev. Letters*. **92**(14), 148102, (2004).