Dynamic neural networks, comparing spiking circuits and LSTM

Arne Koopman, Matthijs van Leeuwen & Jilles Vreeken

Adaptive Intelligence Laboratory, Intelligent Systems Group,
Institute for Information and Computing Sciences, Utrecht University
Correspondence e-mail address: jvreeken@cs.uu.nl

We have investigated two specific network types in the class of dynamic neural networks: LSTM and spiking neural networks. Dynamic neural networks in general are computationally powerful and very promising for tasks in which temporal information has to be processed. We'd like to remark that this is the case for virtually any task or application interacting with the real world. We have tested the networks on a broad set of dynamic tasks and most problems were solved by both; there are some fields though where either LSTM or the spiking neural networks performed better. These differences can be largely brought back to the differences between second and third generation networks.

We have investigated classes of neural networks that are capable of having an internal memory state, i.e. the network just receives input from '*now*' and has to store that somehow in order to influence later outputs. This is a feature required to solve dynamic tasks, tasks in which there is an input-flow that has to be processed without having the help of receiving older inputs again from the outside. Nearly any task in the real world requires such a mechanism, as inputs are typically only received just once.

The neural structure known as once brain will have to temporarily store information in order for you to have a short term memory: a form of memory for which connections between neurons don't need to be physically altered, information is retained by recurrent activity between neurons. We call these dynamic neural networks, of which we'll discuss two quite different types in particular: long-short term memory and spiking neural networks.

Artificial neural networks have become a standard tool within computer science; the first ideas and models are over fifty years old. The first generation of artificial neural networks consisted of McCulloch-Pitts threshold neurons [5], a conceptually very simple model: a neuron sends a binary 'high' signal if the sum of its weighted incoming signals rises above a threshold value. Second generation neurons do not use such a threshold but a continuous activation function to compute their output signals, making them suitable for analogue in- and output. Examples of commonly used activation functions are the sigmoid and hyperbolic tangent. Typical examples of neural networks consisting of neurons of these types are feed-forward and recurrent neural networks.

Real neurons have a base firing-rate (an intermediate frequency of pulsing) and continuous activation functions can model these intermediate output frequencies. Hence, neurons of the second generation are more biologically realistic and powerful than neurons of the first generation [26]. Also, real neurons use individual pulses as signals, short voltage spikes that excite connected neurons. Neuron models of the first two generations do not employ these; for sake of simplicity their output signals are typically single analogue values between 0 and 1. These signals can be seen as normalised firing rates (frequencies) of the neuron. This is a so-called rate coding, where a higher average rate of firing correlates with a higher output signal. Due to such an averaging window mechanism the output value of a neuron can be calculated in iteration. After doing such a cycle for each neuron the response of the network to the input values is known.

In nearly all real-world-related tasks you need to take previously experienced inputs into account in order to determine the appropriate action or conclusion. In other words: the network needs to have some form of memory. Standard feed-forward networks do not have this capability, and without tricks they cannot be used to infer temporal relations. A widely used trick is to present the network not only the current input, but also a window of previous inputs [8,9,17]. This solution is clearly not biologically plausible and has some major disadvantages: only temporal relations within the input-window can be detected and huge input windows are required for long term influences, overtaxing both the system and learning capabilities [14].

Recurrent sigmoid neural networks

Second generation neurons are computationally less complex than their biologically more plausible spiking counterparts and were therefore more appealing in early research, where they were used in various recurrent network topologies [7,11,15]. For example, Elman [7] proposed nets in which an extra recurrent hidden layer, called context units, is trained to trigger the network for specific events. These nets are able to learn tasks like the temporal XOR problem and several grammatical problems, but since they have memories of only a few time steps at most, they aren't capable of dealing with time sequences with long time lags. Extending this memory gave rise to fundamental problems during the training phase of sigmoid recurrent networks. Popular training algorithms for recurrent neural networks include Back-Propagation Through Time (BPTT) and Real-Time Recurrent Learning (RTRL) [9,10,12]. During the learning phase, BPTT gradually enfolds each layer of the network into a multi-layer network, in which each layer represents a snapshot of the corresponding time step. The resulting network allows the error to flow in time and is used for learning temporal correlations. The temporal error is provided in a way similar to that of the well known back-propagation algorithm [29]. A major drawback of BPTT is its need to record the whole network state, inputs, target vectors and weights during the training phase, as weight adjustment is done only after the epoch has ended. In contrast, RTRL allows for real-time weight adjustments, at the cost of losing the ability to follow the true gradient, which gives no practical limitations though [9].

To operate correctly with sigmoid networks, these algorithms require that time lags between inputs and target outputs are kept small; training becomes impossible otherwise. In second generation networks, large time lags tend to either blow the error flow up or let it vanish to zero; leading, respectively, to oscillating weights or a situation where learning does not take place at all. Several solutions to this problem of decaying error flow have been proposed [15,16], from which we have selected Long Short-Term Memory as the second generation alternative for our experiments.

Long Short-Term Memory

An efficient method of dealing with decaying error flow is Hochreiter's Long Short Term Memory (LSTM), of which Constant Error Carrousels (CECs) are an essential element. Their basic function is to ensure a constant error flow by producing the sum of its previous and current inputs (see fig. 1). The model is explained in more detail in Hochreiter's work [16].

Because the error flow does not suffer from decay, interactions with the outside world have to be selected with care: useful error signals have to sustain in the network and irrelevant memory content may not disrupt the current output. Especially with long time lags, time sequences potentially contain a lot of junk input, which harness the useful memory content and therefore does not benefit the learning process.

Restraining this unwanted flow is done by additional regulating gate units that scale the flow from and to the CEC. Gate units receive their input from the input, output and current network state and are trained like normal sigmoid cells to produce the scale factor. In this fashion, gate units can be trained to be selective for certain temporal events and allow the CEC to accumulate the flow of different events. The combination of input gate unit, CEC and output gate unit forms a memory (see fig. 1) cell and is able to satisfy above needs.

When the temporal sequences contain more complex spatial relations at certain time steps, it can be convenient to combine several memory cells together and give them the same temporal selectivity, which makes them focus at the same moment. This is done by grouping several memory cells together that share input and output units to form a memory block.

These memory blocks are integrated into a standard LSTM network topology, in which the input and output layer consists of sigmoid units. The memory blocks reside in the fully connected hidden layer and are optionally aided by sigmoid hidden units.

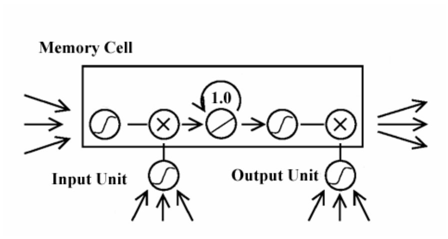

Figure 1. The core of the LSTM network: the Memory Cell with in its centre the *Constant Error Carousel* (CEC), which ensures the constant error flow needed for learning long time dependencies [16].

Training LSTM

The training of LSTM networks is done by an altered version of Real-Time Recurrent Learning, which we shall explain more formally here. Assume that we have a LSTM network, as described above, with n units and m external input lines. Let $y(t)$ denote the n-tuple of outputs of all units at time t, and $x(t)$ the concatenation of m external input signals from the input units I and the output signals from the output units U:

$$x_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases} \tag{1.1}$$

The net input of each unit at time step $t+1$ is very straightforward, and is given by,

$$s_k(t+1) = \sum_{l \in U \cup I} w_{lk}(t) x_l(t) \tag{1.2}$$

This function is used to determine the output of a unit at time step $t+1$. The unit's activation function f_k squashes the net output in a limited range.

$$y_k(t+1) = f_k(s_k(t+1)) \tag{1.3}$$

Since we are dealing with temporal patterns, the target values of output neurons can vary in time. At every time step, all target output units, the set $T(t)$, must meet their target values d. The discrepancy between target value and actual output leads to the current error,

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{if } k \in T(t) \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

This derived error can be used to update the weights, that are adjusted according to their contribution of the current error p and is generally scaled by the overall learning rate, α.

$$\Delta w_{ji} = \alpha \sum_{k \in U} e_k(t) p_{ij}^k(t) \tag{1.5}$$

The influence of the weight w_{ij} on output unit k at time step t is intuitively given by the following gradient,

$$p_{ij}^k(t) = \frac{\partial y_k(t)}{\partial w_{ji}} \tag{1.6}$$

and is derived by tracing down the grown network recursively. This contribution is determined by the current output and the previous outputs of the preceding units: for each output unit k, at time step $t+1$, all ancestor units l, are followed to time step t. Using the Kronecker delta, δ_{ik}, the input is added to this contribution. For each passed unit the output is calculated, which determines the contribution to the current error.

$$p_{ij}^k(t+1) = f_k'(s_k(t+1)) \left[\sum_{l \in U} w_{lk} p_{ij}^l(t) + \delta_{ik} x_j(t) \right] \tag{1.7}$$

We omitted a large part of the complete algorithm, we refer to the work of Williams et al. [9,10] for more details. The LSTM networks in our experiments are trained by a truncated variant of RTRL, which compensates for the

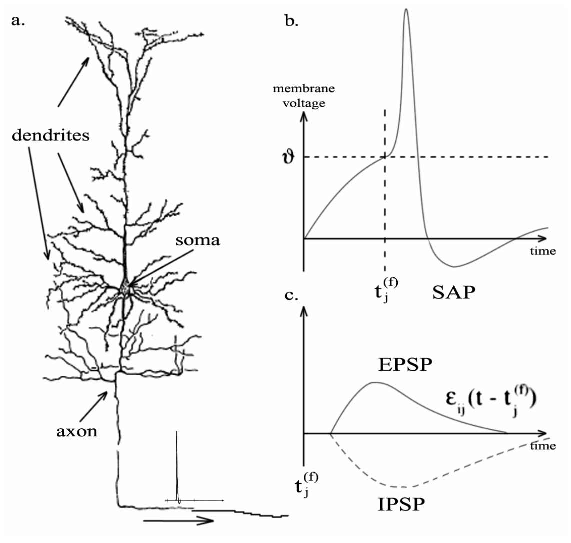

Figure 2. (a) Schematic drawing of a neuron. (b) Incoming post-synaptic potentials alter the membrane voltage so that it crosses threshold value ϑ; the neuron spikes and goes into a refractory state. (c) Typical forms of excitatory and inhibitory postsynaptic potentials over time. [2]

multiplicative dynamics caused by the input and output gates. Upon entering the memory cell, the error signal is scaled by the output unit and can flow through the CEC indefinitely. When it leaves the CEC, it is first scaled by the input unit, used to adjust the incoming weights and is finally truncated. In short, error signals which arrive at a memory cell do not get propagated back further in time.

Spiking neural networks

In the third generation of neural networks, the level of biological realism and computational power is raised by using individual spikes. Spiking neurons are inherently dynamic as they have an ever-changing internal state: their membrane voltage. This provides the network with an internally continuous memory, allowing it to incorporate spatial-temporal information in communication and computation, like real neurons do [6,25]. So instead of using rate coding, these neurons use pulse coding: mechanisms where neurons receive and transmit individual pulses, allowing multiplexing of information as frequency and amplitude of sound [2].

Recent discoveries in the field of neurology have shown that neurons in the cortex perform analogue computations at incredible speeds. Thorpe et al [6] demonstrated that humans analyse and classify visual input (i.e. facial recognition) in under 100ms. It takes at least 10 synaptic steps from the retina to the temporal lobe; this leaves about 10ms of processing-time per neuron. Such a time-window is much too small to allow an averaging mechanism like rate coding [6,4]. This does not mean that rate coding is never used, though when speed is an issue pulse coding schemes are favoured [5].

There are many different schemes for the use of spike timing information in neural computation. We've chosen to use the spike response model, a model in the threshold-fire class of spiking neuron. It's a conceptually simple, easy to implement model that captures key elements of the biologically very realistic Hodgkin-Huxley model [2,3]. We'll cover the details of this model here, further on in this paper we will describe the adaptations we've made in our implementation.

All action potentials are look-alikes. We can therefore forget about their form and characterise them by their firing times $t_i^{(f)}$. The lower index *i* indicates the neuron, the upper index *f* the number of the spike. We can then describe the spike-train of a neuron as

$$F_i = \{t^{(1)}, \dots, t^{(n)}\} \tag{2.1}$$

The variable u_i is commonly used to refer to the internal state, or membrane potential, of a neuron *i*. If a neuron's membrane potential crosses threshold value ϑ from below, it generates a spike. We add the time of this event to F_i, defining this set as

$$F_i = \{\, t \mid u_i(t) = \vartheta \wedge u_i'(t) > 0 \,\} \tag{2.2}$$

When a neuron generates an action potential, the membrane potential suddenly increases, soon followed by a long lasting negative after-potential (see fig. 2b). This sharp rise above the threshold value makes it absolutely impossible for the neuron to generate another spike and is named absolute refractoriness. In the period of relative refractoriness, which we call the negative spike after-potential (SAP), it is less likely that the neuron fires again. We can model this absolute and negative refractoriness with kernel η:

$$\begin{aligned} \eta(s) = & -n_0 \exp\left(-\frac{s-\delta^{abs}}{t}\right) H(s-\delta^{abs}) \\ & - K H(s) H(\delta^{abs} - s) \end{aligned} \tag{2.3}$$

$$H(s) = \begin{cases} 1 \text{ if } s > 0 \\ 0 \text{ if } s \le 0 \end{cases} \tag{2.4}$$

The duration of the absolute refractoriness is set by δ^{abs}, during which large constant *K* ensures that the membrane potential is vastly above the threshold value. Constant n_0 scales the duration of the negative after-potential. Having a description of what happens to a neuron when it fires, we need one for the effect of incoming postsynaptic potentials.

$$\varepsilon_{ij}(s) = \left[\exp\left(-\frac{s-\Delta^{ij}}{\tau_m}\right) - \exp\left(-\frac{s-\Delta^{ij}}{\tau_s}\right) \right] H(s-\Delta^{ij}) \tag{2.5}$$

In equation 2.5, Δ^{ij} defines the transmission delay (axons and dendrites are fast, synapses relatively slow) and $0<\tau_s<\tau_m$ are time constants defining the duration of the effect of the postsynaptic potential. We use variable w_{ij} to model the synaptic efficacy or weight; with which we also can model inhibitory connections by using values lower than zero.

Neurons of the second generation work in the iterative, clock-based manner of digital computers, but can deal with analogue input values; we can quite easily feed input neurons with digitised values from a dataset or a robot-sensor. We cannot just insert such values into a spiking neuron and we will have to affect the membrane-voltage directly according to these values. This is done by $h^{ext}(t)$ that describes all external influences to the neuron's membrane potential.

$$h(t) = h^{ext}(t) + \sum_{j \in \Gamma_i} \sum_{t_j^{(f)} \in F_j} w_{ij} \varepsilon_{ij}(t - t_j^{(f)}) \tag{2.6}$$

The neuron might get excited due to outside influences and fire, effectively transforming an analogue input value into the signal the network can process: a spike. The current excitation of a neuron is described by

$$u_i(t) = \sum_{t_i^{(f)} \in F_i} \eta_i(t - t_i^{(f)}) + h(t) \tag{2.7}$$

where the refractory state, effects of incoming postsynaptic potentials and external events are combined. Together with equation 2.3 this forms the spike-response model, a powerful though easy to implement model for working with spiking neural networks.

Synaptic plasticity

Electrochemical action potentials cannot just jump across the synaptic gap (see fig. 2a) between two neurons; a post-synaptic potential (see fig. 2c) has to be induced by chemicals (neuro-transmitters) that travel across the gap. This involves many variables, like the amount of readily deployable vesicles of neuro-transmitters and the capability to replenish those. All these variables affect the resulting post-synaptic potential. The synapse is therefore no simple signal transferring device, but a highly complex signal-pre-processor. Due to these effects, synapses play an important role in development, memory and learning of neural structures. Synaptic plasticity is a form of altering the pre-processing, which is a preferred word for '*learning*' as it better describes what is at hand: long- or short-term change in synaptic efficacy [1,4,6].

Hebbian plasticity is a local form of long-term potentiation (LTP) and depression (LTD) of synapses and is based on the correlation of firing activity between pre- and postsynaptic neurons. This is usually, and easily, implemented with rate coding: similar neuron activity means a strong correlation. As we use a pulse-coding scheme now, we have to think about how to define correlations in neural activity using single spikes. Pure Hebbian plasticity acts locally at each individual synapse, making it both very powerful and difficult to control; it is a positive-feedback process that can destabilize postsynaptic firing rates by endlessly strengthening effective and weakening ineffective synapses. If possible, one has to avoid such behaviour, most desirably by a biologically plausible local rule.

Spike-timing dependent synaptic plasticity (STDP) is a form of competitive Hebbian learning that uses the exact spike timing information [1]. Experiments in neuroscience have shown that long-term synaptic strengthening occurs when presynaptic action potentials arrive within 50ms before a postsynaptic spike and weakening when it arrives late. Due to this mechanism STDP can lead to stable distributions of LTP and LDP, making postsynaptic neurons sensitive to the timing of incoming action potentials. This sensitivity leads to competition among the presynaptic neurons, resulting in shorter latencies, spike synchronization and faster information propagation through the network [1].

Hebbian plasticity is a form of unsupervised learning, which is useful for clustering tasks but less appropriate when a desired outcome for the network is known in advance. Back-propagation [29] is a widely known and often used supervised learning algorithm. Due to the very complex spatial-temporal dynamics and continuous operation it cannot be directly applied to spiking neural networks, adaptations [18] exist in which individual spikes and their timing are taken into account.

However, spiking backprop suffers from some disadvantages. The main objection is that networks need to be overly complex: instead of optimising the single delay of a synapse, the algorithm needs a range of varying delayed connections between neurons. A second argument against this particular algorithm is that it can only work with single-spike correlations, learning to detect temporal patterns or other temporal tasks are not feasible. Still, spiking backprop is currently the only supervised learning algorithm available for networks of spiking neurons.

Artificial evolution

Genetic algorithms [22] are widely used for optimisation and search problems, in particular when the parameter-space to explore is extremely large. When applied to general search problems, a large set of possible solutions is evaluated and recombined to form a new generation of possible solutions. At the start of an experiment we use a randomly initialised population, consisting of a certain number of individuals. Each individual encodes a controller, which should be read here as 'anything that accepts input values and gives output values after processing'. The performance ('fitness') of each individual can be measured by testing the controller on the given task.

An individual consists of a genotype and a phenotype. The genotype is the genome, in the simplest case a binary string of fixed length. Genomes can be far more advanced though: multiple chromosomes, a larger DNA alphabet, variable string lengths, etcetera. When an individual's performance is tested, the first thing to do is to build the phenotype corresponding to the genotype, which can then be used for the specified task.

When artificial evolution is applied on robots, it is called evolutionary robotics [24,27,28]. In this case, each individual of a population is tested on a real robot: the objective is to evolve controllers that are capable of controlling a real robot in the real world. In our experiments, the controllers were spiking neural networks and used for more static tasks: fixed time series were given (consisting of both input and target output) for which the objective was to evolve controllers that could approach the target output as much as possible.

After determining fitness values for all individuals in a population, reproduction can be done. There are several ways for doing this, we've used truncation selection: keep only the best individuals for reproduction, dispose of the rest. Very important characteristics of evolution are the genetic 'operators': crossover and mutation. We used basic single-point crossover, cutting the parent DNA strings at the same point and switching the ends. The mutation operator toggles every single bit of the genome with a certain chance. For improved evolutionary stability, elitism was used in reproduction: by always retaining the best individual (without modifying the genome) we ensured that our search wouldn't loose the current best solution.

The tasks

Our goal was to compare a second generation with a third generation neural network type on dynamic tasks. We therefore composed tasks that require the networks to do more than statically map single input values to single output values. In other words, an internal state or history of previous inputs is required to be able to produce the correct output.

To the best of our knowledge, evolving spiking neural networks for such time series tasks has hardly been done before. Because of this we weren't sure what performance to expect and thus started with a few simple tasks. We will now describe all tasks that we created data sets for, which we used both to evolve spiking neural networks and to train the LSTM networks with.

Frequency detection. The goal is to classify four different 'frequencies' (fire rates) that are fed into the network. There is one output per frequency to classify and this should be 1 when the corresponding frequency is detected, 0 otherwise. To make sure the detection isn't based on integration, the integrals of the different frequency parts are equal. An extra input is provided to indicate the start of a new frequency block and request output of the previous block (output at other moments doesn't influence fitness).

Gradient. In this task the network was asked to classify the direction of a gradient: the network had to determine whether the gradient was positive (increasing input values) or negative (decreasing input values). In other words, the network had to detect the sign of the first derative.

Inverse binary. A rather simple task: series of 0's and 1's are given as input, the network should output the opposite. Thus, 0 gives 1 and 1 gives 0.

Inverse continuous. A more advanced version of the previous task, this continuous inverse also requires the networks to output the inverse of the input, but the input is now a continuous value between 0 and 1. A simple formula that describes this behaviour is $out(t) = 1 - in(t)$ (where t is the current time step).

Memory. In this very difficult task, the network has to repeat a previously seen input on command. First, either 0 or 1 is given as input for some time, after which a period of no input follows. Once the second input line signals by switching to 1, the originally seen input should be given as output. Before this, the output is unimportant and doesn't influence the fitness value.

Sines. A difficult classification task. Networks are asked to classify two types of sines, where the frequency is equal, but the amplitude is scaled with either 0.5 or 1.0.

Switch. A task where an internal state is an absolute necessity. We tried two versions, in which the input-line has a base value of 0 or 1. The initial desired output value is always 0, and has to be kept so until the input line indicates a switch. This is done by a short (1 time step) peak (i.e. from 0 to 1, or vice-versa). The output has to be kept at 1 until the next switch signal, etcetera. Summarising, each cycle the input is the inverse of the base value; the output value should be inversed (switched).

Temporal XOR. One single input value is randomly chosen every time step and is either 0 or 1. The corresponding output should be equal to $in(t)\ xor\ in(t-1)$. In other words, the XOR of the last two inputs should be given as output.

We will give more details on the specific data sets we used in our experiments in the section on the results.

Evaluating Long Short-Term Memory

For our experiments we have used the original software written by Hochreiter [13,16] and slightly adjusted this in order to suite our purposes. Training and evaluation is done on different data sets, which were made compatible with the spiking circuits data sets by keeping data in the range of the interval [0,1]. In order to easily generate datasets incorporating random perturbations for LSTM datasets, a data generation program, which generated data for both LSTM and spiking circuits, was used.

We have briefly discussed the learning algorithm used with LSTM and proceed with the evaluation of the network. Evaluating the network is done by applying an input from the test set and measuring the error between output and target, this resulting total error is the summed squared error of all output cells for the duration of all sequences.

$$\text{sequence error} = \sum_t \sum_{k \in U} (d_k(t) - y_k(t))^2 \qquad (3.1)$$

Training is finished when either the maximum amount of training period is reached or the error has reached the minimum specified value. After training the network is tested on the test dataset and its output is dumped to a file. This procedure is repeated for a given number of trials.

Evolving spiking circuits

In order to use the spike response model for artificial evolution, we applied some simplifications to this standard model in order to avoid overly large genomes and limiting the amount of computation needed. Our derivation of the model is based mainly on the model as described by Floreano and Mattiussi [23]. The software we used is based on *i* [28], an application written for evolutionary robotics with spiking circuits. We started with this application and developed it further to suit our needs.

A first important note is that time has been made discrete: continuous time is assumed in the spiking response model, but discrete time steps were introduced to make implementation easier. Neurons are updated each time step, but input and output are respectively set and read only at specified intervals (read further for details).

A spike is assumed to last one time step, after which one time step of absolute refractoriness holds. After this, the refractory kernel is computed for each neuron using the simplified formula:

$$\eta(s) = -\exp\left(-\frac{s}{\tau_m}\right) \qquad (3.2)$$

Here, s is the number of time steps since the last spike time of this particular neuron, as we take into account only the refractory kernel of the last spike for each neuron. The formula for postsynaptic contributions of incoming spikes (2.5) remains the same, although only incoming spikes of the last 20 time steps are taken into account.

We now have to describe the architecture of the networks and the handling of in- and output, as both of these are unconstrained in the general model. We distinguish receptor neurons and interneurons: receptor neurons are used for feeding input into the network and only have outgoing synaptic connections; interneurons may have incoming and outgoing synaptic connections and may be fully interconnected and recurrent. The number of receptor neurons is given by the task; the number of interneurons is at least the number of required output values, as one interneuron is used for each output. For some tasks, we noticed that it was necessary to add one additional receptor neuron with a constant input of 1 to facilitate basic network activity.

In our experiments, all neurons (receptor and interneurons) in the network were updated each time step, but a certain number of these updates together form a cycle. The number of updates per cycle is fixed within an experiment, but we changed this resolution between experiments. At the first update of a cycle, new input values (in the range [0,1]) are fed into the receptor neurons. At each update, each receptor neuron stochastically determines its $h^{ext}(t)$, based on the current input:

$$h^{ext}(t) = flip(input) \qquad (3.3)$$

flip() returns 1 with a chance equal to its argument, 0 otherwise. As a membrane potential of one crosses the threshold, this would normally gives a spike (but please note that receptor neurons also have a refractory period, the maximum fire rate for all neurons is 0.5).

It is clear that fire rate coding is used to encode the input; the same is used to determine the output of the network. For each interneuron that is used for output, all spikes during the last so-many updates of a cycle are counted and divided by the maximum number of spikes that could have occurred, giving a value properly scaled between 0 and 1. The number of updates at the end of each cycle that contribute to the output value is varied between the experiments, just like the number of updates per cycle. It is obvious that a higher resolution output also requires a larger number of updates per cycle.

Although we used evolution to search for network parameters as much as possible, we chose some parameters fixed to avoid ending up with huge genomes and being unable to evolve anything useful. Let us first discuss the fixed parameters before proceeding with the genotype-phenotype mapping. The threshold ϑ was set to 0.5, which means that only a few recent incoming spikes are necessary to evoke a spike. The synapse and neuron time constants, τ_s and τ_m, are set to 10 and 4, respectively.

We now arrive at the genotype-phenotype encoding. The genome, a binary string, consists of a certain number of blocks, each block encoding one spiking interneuron. The first bit of each block encodes whether the interneuron is excitatory or inhibitory: a spike coming from an excitatory neuron adds to the membrane potential, vice versa for inhibitory neurons. The remaining bits of each block encode properties of incoming synapses; enough bits are present to encode for synapses coming from all receptors and interneurons. One could also imagine dividing these bits into blocks: one block per synapse, the first bit encoding for synaptic presence (i.e., whether there is a synaptic connection or not), the next four bits encoding synaptic strength (scaled between 0 and 2) and another four bits encoding the synaptic delay (between 0 and 15).

Using this encoding, each population of *n* individuals encodes *n* spiking circuits, which can all be built and tested on a given task. As we described, each task consists of a number of input and target output values that belong together in a fixed order. When a certain input value is fed into the circuit, the output should be as near to the target output as possible. As we are dealing with time series here, it's often not enough for a circuit to base the output on the current input only: an internal state has to be used in order to perform well.

The number of cycles that each network was tested for a task depended on the task, but the fitness function was always the same, based on the difference between actual and target outputs,

$$fitness(t) = 1 - \sum_x \sqrt{|out_x(t) - tar_x(t)|} \qquad (3.4)$$

where *fitness(t)* is the fitness value at cycle *t*, $out_x(t)$ and $tar_x(t)$ are the actual and target output values, respectively, of neuron *x* at cycle *t*. The fitness values of all cycles are summed and divided by the total number of cycles to normalize between 0 and 1. An output value was not required for every input; for a few tasks, target output was only given for some pre-defined intervals. Output values outside these intervals didn't influence the fitness.

The evolution parameters used for the experiments varied: for particular tasks the population size was changed between 60 and 500, while the default size was 120. For truncation selection, the best 25% of the population was always selected for reproduction. The crossover and mutations rates were 0.1 and 0.05, respectively. The maximum number of generations was 300, but less generations were enough in most cases.

Results

We have tested both types of networks, using the techniques described above, on the temporal tasks that were explained earlier. Thus, we trained LSTM networks for these tasks and evolved spiking circuits for exactly the same tasks, to enable us to make a comparison.

Our LSTM network topology consists of one input unit, one output unit and no additional conventional hidden units. The hidden layer consisted of two memory blocks, each with 2 memory cells, which was enough for most of the experiments we conducted, and the learning rate was set to 0.1 by default.

Most of the parameters for artificial evolution and spiking circuits have already been described, but there are a few parameters that haven't been settled yet. Almost all used data sets consist of 200 cycles, making evolution quite fast:

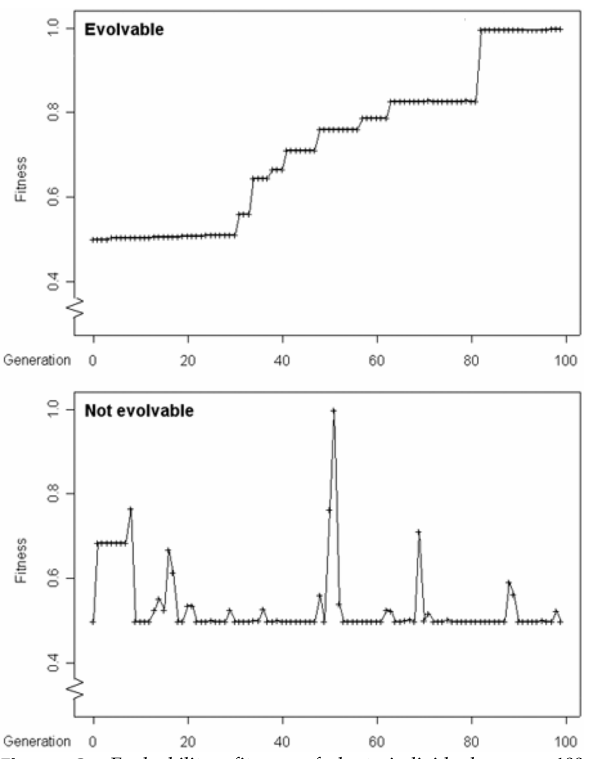

Figure 3. Evolvability, fitness of best individuals over 100 generations. *Top:* Typical increasing fitness value, the task is evolvable. *Bottom:* The maximum stays at the same low value and higher values are just lucky individuals, as the fitness drops back again: not evolvable.

each individual only needs to be tested for 200 cycles (1 epoch), as results turned out to be constant when more epochs were used. Only the frequency detection data set was significantly larger: 800 cycles.

All tasks were evolved with 5 interneurons in total, we experimented with both more and less interneurons, but more didn't give much better results and less wasn't always enough. For other (task-dependent) settings, we refer to Table 1. We first tried all tasks with the default settings (40 neuron updates per cycle and 20 neuron updates to determine output values), but had to change this in a few cases (essentially to increase resolution). We will now describe the results we obtained for each task separately.

Frequency detection. Evolution is very good at finding simple strategies that get a relatively high fitness, but these strategies are not always in accordance with the objective. This is also the case for the frequency detection task: two frequencies seem to be recognised and that gives already fairly good fitness, but the other frequencies are ignored and that wasn't the purpose of the task. LSTM also seems to have problems with this task; it only discriminates between the signals containing pulses and the low threshold input, surely not the preferred behaviour.

Gradient. We had to use two different data sets for LSTM: in the original data set, the slopes ended at different values (i.e. no simple ascending from 0 to 1) and this produced unpredictable results. Instead of providing a classification, the output was the inverse of the input (a very surprising result). In the second dataset, all sequences end at the same value (i.e. ascending or descending to 0.5) and LSTM networks are able to classify correctly. We had no such oddities with evolution of spiking networks, but no good individuals were evolved whatsoever. A commonly found strategy was to give high output when the input is high and keep it that way for some time, this turned out to work well. (Similar strategies were found when we tried other data sets.)

Task					Spiking circuits				LSTM
Name	#in	#out	Target?	State?	Bias?	#updates	#updOut	SSE	SSE
Frequency detection	2	2	Some	Yes	No	40	20	0.250	0.2684
Gradient	1	1	All	Yes	Yes	40	20	0.182	0.0262
Inverse binary	1	1	All	No	No	40	20	0.000	0.0742
Inverse continuous	1	1	All	No	No	100	80	0.003	0.0101
Memory	2	1	Some	Yes	Yes	40	20	0.500	0.1111
Sines	2	1	Some	Yes	Yes	80	20	0.455	0.2954
Switch	1	1	All	Yes	Yes	40	20	0.118	0.8914
Temporal XOR	1	1	All	Yes	Yes	40	20	0.250	0.4989

Table 1. Overview of all results. Properties of each task are given, also some spiking circuit settings and the sum squared error of both types of network. Task properties: #in = number of input values, #out = number of output values, Target? = target output defined for?, State? = internal state required to accomplish task. Spiking circuit settings: Bias? = bias receptor added?, #updates = number of neuron updates per cycle, #updOut = number of updates used to determine output value.

Inverse binary. LSTM could only produce viable results when the duty cycle was raised to 0.5. In that case, the trained network was on par with our evolved circuits that were perfect solutions. The LSTM networks were just slightly less perfect, as the network always needed a cycle to adjust it's output to the changing input. Spiking circuits didn't need this, they gave the correct inverse even when the input was randomly chosen between 0 and 1 each cycle.

Inverse continuous. Performance of the LSTM network was equal to that of the previous task: again the duty cycle had to be raised. The evolved spiking circuits did fairly well again also, but the resolution of input and output is a bottleneck here. As we are working with a (discrete) number of spikes each cycle and not with continuous numbers (as LSTM), it is very important that input and output resolution are in accordance with the number of neuron updates each cycle.

Memory. This task shows us the profound advantages of learning over evolutionary search: LSTM can learn this task without too much effort, while our evolutionary approach with spiking networks is unable to reproduce the previously seen input when requested.

Sines. This task proved to be too difficult to be solved by evolution as we used it. Even though fitness reached 0.75 at various attempts, the behaviour of the network is far from right and it cannot classify the input sine waves. As for LSTM, classifying the sines fails completely.

Switch. The results that we obtained with this task gave us an interesting difference between the two network types. The spiking neural networks found by evolution shows nearly perfect behaviour, only suffering from the fact that it cannot switch its state immediately: it needs two cycles to complete its output change. The same lagging behaviour was seen in the LSTM network, but certainly not with the proposed data set: a 1-cycle input signal was insufficient to switch the output for all topologies tested, the networks simply kept their output at 0. It was not until we lengthened this signal to half (!) of the sequence's length, that the network showed behaviour more like that of the evolved spiking circuit.

Temporal XOR. This (unavoidable) XOR-task posed serious problems for both of our approaches. We did not succeed in successfully evolving a spiking neural network capable of solving the described task. All our evolutionary runs (partially with different spiking circuit parameters) came up with an efficient solution of fitness 0.75: the output is always high, except after two subsequent zeroes. None of the many tested LSTM topologies could find a solution for the temporal XOR task as described. Output and error remained around the 0.5 during runs after learning. An efficient solution, but not quite what we were after, was found by imposing a delay (only giving 0 as input) after each offered input pair to be XORed.

Comparison

The different tasks give widely varying results for the types of networks experimented with: some tasks can be solved by both without too much effort, but this isn't the case for all tasks and some turned out to be infeasible with the parameters and techniques we used.

No serious problems were encountered with the two inverse problems, for which no internal state was required and feed-forward (non-recurrent) neural networks could also be used. As already mentioned, the resolution of input and output is an important issue here and that's something that counts for many real tasks: using rate coding in spiking circuits make that only a certain amount of detail can be dealt with, LSTM doesn't have this problem because it deals with analogue values internally. If very little differences in input (or output) make large differences in a task, it may be more straightforward to use a second generation network like LSTM. Another possibility is to try pulse coding schemes with the spiking circuits (e.g. spike time coding) to make encoding input and output values more precisely.

Too difficult for both network types were the sine classification, temporal XOR and frequency detection tasks, but these should be investigated further: we think that especially spiking circuits could perform better if we improved them by adding synaptic plasticity. Evolution is good at finding simple strategies to increase fitness, but these tasks were too difficult to evolve. Individuals that obtained a higher fitness were just lucky, not better at the task at hand. Evolvable tasks show an increasing maximum fitness during an evolutionary run, runs with too difficult tasks show a more or less constant fitness (see fig. 3).

LSTM performed better than spiking networks at two tasks: the gradient sign detection and memory tasks. Evolution was unable to find suitable spiking networks for these, which is not surprising for the memory task: a long time relation between input and output has to be found, basically by coincidence. That the gradient sign detection also gave problems may possibly be attributed to the stochastic rate coding: it may be difficult to accomplish this when the gradient is low and the stochastic receptors inflict even more noise in the spike trains.

The one task that spiking circuits were better at than LSTM was the switching task. LSTM networks are unable to completely revise their internal state based on one single input, whereas this is no problem for spiking networks: the neurons are updated 40 times each cycle and a one-cycle change of the input can have a large impact on the internal state of the whole network. This change wasn't always finished within one cycle, but the best networks completed the switch even after the input was back to normal in the next cycle.

Discussion

Neural structures as found in nature are very well suited for the processing of temporal information: these networks have an internal dynamic memory state that may be influenced for a shorter or longer time by its inputs – long and short term memory. This quality is not found in most classes of artificial neural networks: only recurrent and spiking neural networks perform well in spatio-temporal domains.

Second generation neurons use analogue values and a continuous activation function to compute their output. Many neural networks have been proposed in this generation, these can largely be divided in feed-forward and recurrent neural networks. As feed-forward networks have no internal state and cannot be easily used for dynamic tasks, we focused on recurrent neural networks. We covered some basics of sigmoidal recurrent networks and mentioned some learning algorithms, BPTT and RTRL that can be used to learn temporal correlations. Furthermore, we explained Long Short-Term Memory, a particular strong type of recurrent neural network, as it doesn't suffer from error flow problems as most others.

Spiking neural networks, incorporating third generation neurons, use the element of time in communicating by sending out individual pulses. Spiking neurons can therefore multiplex information into a single stream of signals, like the frequency and amplitude of sound in the auditory system [6]. We have covered the very general and realistic spike-response model, a powerful and realistic model for using pulse coding in neurons. Standard neural network training algorithms use rate coding and cannot be directly used satisfactory for spiking neural networks, therefore we have used evolution to find suitable network topologies and parameters.

We have chosen two specific network types, one from each network generation, and have tested them on a number of dynamic tasks. Some tasks proved too difficult, some were no problem for both networks. There are some fields though where either LSTM or spiking circuits performed better. The difference can be largely brought back to the differences between second and third generation networks. LSTM is an architecture combined with a learning method that is aimed at finding temporal correlations and working with analogue values. Using so-called forgetting gates [30] with LSTM might improve the performance on the more difficult tasks. Spiking circuits work with individual pulses and evolving network properties is a very different way of finding solutions and is not always good enough, which we have shown. But although it is difficult to improve much on LSTM, there is much work to be done on spiking neural networks. Spike-timing dependent synaptic plasticity uses exact spike timing to optimise information-flow through the network, as well as it imposes competition between neurons in the process of unsupervised Hebbian learning. We think such a form of learning would be very beneficial for spiking circuits and could make it possible to find solutions for the more difficult tasks.

Dynamic neural networks in general are computationally powerful [2,3,4,5,6] and very promising for tasks in which temporal information plays an important role. We'd like to conclude remarking that this is the case for virtually any task or application that has interaction with the real world.

References

1. Abbot, L. F. & Nelson, S. B. *Synaptic Plasticity: taming the beast*, Nature Neuroscience Review, vol. 3 p.1178-1183 (2000).
2. Gerstner, W. *Spiking Neurons* in Maass, W. & Bishop, C. M. (eds.) *Pulsed Neural Networks*, MIT-press (1999).
3. Gerstner, W., Kistler, W. *Spiking Neuron Models*, Cambridge University Press (2002).
4. Maass, W., Schnitger, G. & Sontag, E. *On the computational power of sigmoid versus boolean threshold circuits*, Proc. of the 32nd Annual IEEE Symposium on Foundations of Computer Science, p. 767-776 (1991).
5. Maass, W. *Synapses as Dynamic Memory Buffers*, Technische Universität Graz (2000).
6. Thorpe, S., Delorme, A., Van Rullen, R. *Spike based strategies for rapid processing*, Neural Networks, vol. 14(6-7), p.715-726 (2001).
7. Elman, J.L. 'Finding Structure in Time'. In: *Cognitive Science*, vol. 14, p.179-211 (1990).
8. Kool, A. *Literature Survey*, Center for Dutch Language and Speech, University of Antwerp (1999).
9. Williams, R.J. & Zipser, D. 'A Learning Algorithm for Continually Running Recurrent Neural Networks'. In: *Neural Computation*, 1, pp.270-280 (1989).
10. Williams, R.J. & Peng, J. 'An Efficient Gradient-Based Algorithm for online Training of Recurrent Neural Network Trajectories'. In: *Neural Computation*, 2, pp.490-501 (1990).
11. Horne, B.G. & Giles, C.L. 'An experimental Comparison of Recurrent Neural Networks', In: Tesauro, G., Touretzky, D. And Leen, T. (eds), *Neural Information Processing Systems 7*, MIT Press, p.697 (1995).
12. Pearlmutter, B.A. 'Gradient Calculations for Dynamic Recurrent Neural Networks: A Survey'. Draft of July 20, 1995 for: *IEEE Transactions on Neural Networks* (1995).
13. Hochreiter, S., Bengio, Y., Frasconi, P. and Schmidhuber, J., 'Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies'. In: Kremer, S. C. and Kolen, J. F. (eds), *A Field Guide to Dynamical Recurrent Neural Networks*. IEEE press (2001).
14. Schmidhuber, J. & Hochreiter, S. *Guessing can Outperform many Long Time Lag Algorithms*, Technical note IDSIA-19-96 (1996).
15. Lin, T., Horne, B. G., Tino, P. and Giles, C. L. 'Learning long-term depencies in NARX recurrent neural networks'. In: *IEEE Transactions on Neural Networks*, vol. 7(6), p.1329 (1996).
16. Hochreiter, S. & Schmidhuber, J. *Long Short-Term Memory*, Neural Computation vol. 9(8), p.1735-1780 (1997).
17. Giles, L., Lawrence, S. and Tsoi, A. C. *Noisy Time Series Prediction using a Recurrent Neural network and Grammatical Inference*, Machine Learning, vol. 44(1-2), p.161-183 (2001).
18. Bohte, S.M, Kok, J.N. & La Poutré, H. *Error-Backpropagation in Temporally Encoded Networks of Spiking Neurons*, Neurocomputing preprint (2000).
19. Gomez, F.J. & Miikkulainen, R. 'Solving non-Markovian Control Taks with Neuroevolution'. In: Dean, T. (ed), *Proceedings of the International Joint Conference on Artificial Inteligence (IJCAI99)*, Denver: Morgan Kaufmann (1999).
20. Pujol, J.C.F. & Poli, R. 'Efficient Evolution of Asymmetric Recurrent Neural Networks Using a PDGP-inspired Two-dimensional Representation'. In: Banzhaf, W., Poli, R., Schoenauer, M. & Fogarty, T. (eds), *Proceedings of the First European Workshop on Genetic Programming (EUROGP)*, Lecture Notes in Computer Science, vol. 1391, Springer-Verlag, p.130-141 (1998).
21. Esparcia-Alcazar, A.I. & Sharman, K. 'Evolving Recurrent Neural Network Architectures by Genetic Programming'. In: Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L. (eds). *Genetic Programming 1997: Proceedings of the Second Annual Conference*. San Francisco, CA: Morgan Kaufmann, pp. 89-94 (1997).
22. Goldberg, D.E. *Genetic Algorithms in Search, Optimization and Machine Learning*, Reading, MA: Addison-Wesley (1989).
23. Floreano, D. & Mattiussi, C.. `Evolution of Spiking Neural Controllers for Autonomous Vision-Based Robots'. In Gomi, T., ed., *Evolutionary Robotics. From Intelligent Robotics to Artificial Life*. Tokyo: Springer Verlag (2001).
24. Nolfi, S. & Floreano, D. *Evolutionary Robotics: Biology, Intelligence, and Technology of Self-Organizing Machines*. Cambridge, MA: MIT Press. 2nd print (2001).
25. Vreeken, J. *Spiking neural networks, an introduction*. Institute for Information and Computing Sciences, Utrecht University (2002).
26. DasGupta, B. & Schnitger, G. *The power of approximating: a comparison of activation functions*, Advances in Neural Information Processing Systems, vol. 5 p.363-374 (1992).
27. Zufferey, J.C., Floreano, D., Van Leeuwen, M. & Merenda, T. 'Evolving Vision-based Flying Robots'. In: Bülthoff, Lee, Poggio, Wallraven (eds), *Proceedings of the 2nd International Workshop on Biologically Motivated Computer Vision*, LNCS, Berlin, Springer-Verlag (2002).
28. Van Leeuwen, M., *Evolutionary blimp & i*. Internship report, Institute for Information and Computing Sciences, Utrecht University (2002).
29. Rumelhart, D.E., Hinton, G.E. & Williams, R.J. *Learning representations by back-propagating errors,* Nature Vol. 323 (1986).
30. Gers, F.A., Schmidhuber, J. & Cummins, F. 'Learning to forget: Continual prediction with LSTM'. In: *Neural Computation*, vol. 12(10) p.2451-2471 (2000).