Long short-term memory and learning-to-learn in networks of spiking neurons

Guillaume Bellec*, Darjan Salaj*, Anand Subramoney*, Robert Legenstein & Wolfgang Maass
Institute for Theoretical Computer Science
Graz University of Technology, Austria
{bellec,salaj,subramoney,legenstein,maass}@igi.tugraz.at
* = first authors

Abstract

The brain carries out demanding computations and learning processes with recurrent networks of spiking neurons (RSNNs). But computing and learning capabilities of currently available RSNN models have remained poor, especially in comparison with the performance of recurrent networks of artificial neurons, such as Long Short-Term Memory (LSTM) networks. In this article, we investigate whether deep learning can improve RSNN performance. We applied backpropagation through time (BPTT), augmented by biologically inspired heuristics for synaptic rewiring, to RSNNs whose inherent time constants were enriched through simple models for adapting spiking neurons. We found that the resulting RSNNs approximate, for the first time, the computational power of LSTM networks on two common benchmark tasks. Furthermore, our results show that recent successes with applications of Learning-to-Learn (L2L) to LSTM networks can be ported to RSNNs. This opens the door to the investigation of L2L in data-based models for neural networks of the brain, whose activity can – unlike that of LSTM networks – be compared directly with recordings from neurons in the brain. In particular, L2L shows that RSNNs can learn large families of non-linear transformations from very few examples, using previously unknown network learning mechanisms. Furthermore, meta-reinforcement learning (meta-RL) shows that LSNNs can learn and execute complex exploration and exploitation strategies.

1 Introduction

Recurrent networks of spiking neurons (RSNNs) are frequently studied as models for networks of neurons in the brain. In principle, they should be especially well-suited for computations in the temporal domain, such as speech processing, as their computations are carried out via events in time and space. One difference between RSNNs in the brain and RSNN models is that RSNNs in the brain have been optimized for their function through long evolutionary processes, complemented by a sophisticated learning curriculum during development. Since most details of these biological processes are currently still unknown, we asked whether deep learning is able to mimic these complex optimization processes on a functional level for RSNN models. We used backpropagation through time (BPTT) for deep learning, which we adapted for RSNNs similar to [1][2]. In order to also optimize the connectivity of RSNNs, we augmented BPTT with a biologically inspired heuristics for synaptic rewiring [3][4]. Compared to LSTM networks, RSNNs tend to have inferior short-term memory capabilities. Since neurons in the brain are equipped with a host of dynamics processes on time scales larger than a few dozen ms [5], we enriched the inherent dynamics of neurons in our model by a standard neural adaptation process.
We first show (section 3) that this approach produces new performance levels of RSNNs for two common benchmark tasks: Sequential MNIST and TIMIT. We then show it also makes L2L applicable to RSNNs (section 4). Finally, we show that meta-RL [6, 7] is also applicable to RSNNs (section 5). This links a recent model for reward-based learning in the brain [8] to biologically more realistic models on the neural network level.

Application of L2L to LSTM networks started quite some time ago, see e.g. [9], and has lead in recent years to a host of remarkable new methods and results, see e.g. [6, 7]. The reason why LSTM networks have been used for most work on L2L for neural networks is that they can naturally accommodate two levels of learning and representation of learned insight: synaptic connections and synaptic weights can encode “innate” knowledge, as well as transfer knowledge from preceding learning of more-or-less related tasks. Even when its weights are fixed, the neural network can use its short-term memory to accumulate knowledge and produce predictions during a single learning task. The process underlying learning resides entirely in the dynamics of the internal state of the recurrent network. It has recently been argued [8] that this is in fact a way how fast learning is implemented in the pre-frontal cortex (PFC). The dopamine system trains the PFC to operate as its own free-standing learning system. Of course, many other schemes for encoding, fusing, and applying results of long-term meta-learning and results from the current learning task are also of interest [10, 11]. The investigation of these new perspectives of network learning is of particular interest in view of debates in neuroscience about the role of synaptic plasticity for learning. In particular, the role of spike-timing-dependent plasticity (STDP) for fast learning [13, 14] is not clear. Lack of evidence for the role of dopamine gated synaptic plasticity for fast reward-based learning in the PFC is addressed in the text to Suppl. Fig. 3 in [8].

Our results show that L2L induces a powerful new method for fast learning in RSNNs. L2L results for RSNNs have the advantage that many features and fingerprints of resulting new learning algorithms for the network, such as neural activity (spike trains), changes in the excitability of neurons and synaptic weights, and changes in the network dynamics through learning can be compared with measurements from networks of neurons in the brain.

At the same time, the resulting new methods for learning nonlinear functions from few examples are also of interest for spike-based neuromorphic chips such as Brainscales [16], SpiNNaker [17], True North [2], chips from ETH Zürich [18], and Loihi [19]. Nonlocal learning rules such as backprop are challenges for some of these neuromorphic devices (and for many brain models), and alternative paradigms for learning nonlinear functions – like the ones we exhibit – are desirable.

2 LSNN model and deep learning methods

The LSNN model: LSNN consists of a population $R$ of integrate-and-fire (LIF) neurons (excitatory and inhibitory), and a second population $A$ of LIF excitatory neurons whose excitability is temporarly reduced through preceding firing activity, i.e., these neurons are adapting (see Suppl.). It is well known that a substantial fraction of excitatory neurons in the brain is adapting, with diverse time constants, see e.g. the Allen Brain Atlas [20] for data from the neocortex of mouse and humans. Both populations $R$ and $A$ receive spike trains from a population $X$ of external input neurons. Results of computations are read out by a population $Y$ of external linear readout neurons, see Fig. 1C.

Common ways for fitting models for adapting neurons to data are described in [21, 22, 23, 24]. We are using here the arguably simplest model: We assume that the firing threshold $B_j(t)$ of neuron $j$ increases by some fixed amount $\beta/\tau_{a,j}$ for each spike of this neuron $j$, and then decays exponentially back to a baseline value $b^0_j$ with a time constant $\tau_{a,j}$. Thus the threshold dynamics for a discrete time step of $\delta t = 1\, \text{ms}$ reads as follows

\begin{align}
B_j(t) &= b^0_j + \beta b_j(t), \\
b_j(t + \delta t) &= \rho_j b_j(t) + (1 - \rho_j) z_j(t),
\end{align}

where $\rho_j = \exp(-\frac{\delta t}{\tau_{a,j}})$ and $z_j(t)$ is the spike train of neuron $j$ assuming values in $\{0, \frac{1}{\rho_j}\}$. Note that this dynamics of thresholds of adaptive spiking neurons is similar to the dynamics of the state of context neurons in [25]. It generally suffices to place the time constant of adapting neurons into the desired range for short-term memory (see Suppl. for specific values used in each experiment).
We refer to the resulting type of RSNNs as Long short-term memory Spiking Neural Networks (LSNNs).

**Deep learning and L2L for LSNNs:** We optimize the synaptic weights, and in some cases also the connectivity matrix of an LSNN for specific ranges of tasks. The optimization algorithm that we use, backpropagation through time (BPTT), is not claimed to be biologically realistic. But like evolutionary and developmental processes, BPTT can optimize LSNNs for specific task ranges. Backpropagation (BP) had already been applied in [1] and [2] to feedforward networks of spiking neurons. In these approaches, the gradient is backpropagated through spikes by replacing the non-existent derivative of the membrane potential at the time of a spike by a pseudo-derivative that smoothly increases from 0 to 1, and then decays back to 0. We reduced (“dampened”) the amplitude of the pseudo-derivative by a factor < 1 (see Suppl. for details). This enhances the performance of BPTT for large time spans, that require backpropagation through several 1000 layers of an unrolled feedforward network of spiking neurons. Another implementation of BPTT for RSNNs was proposed in [26]. In order to optimize not only the synaptic weights of a RSNN but also its connectivity matrix, we integrated BPTT with the biologically inspired [3] rewiring method DEEP R [4] (see Suppl. for details). DEEP R converges theoretically to an optimal network configuration by continuously updating the set of active connections [27, 3, 4].

The standard setup of L2L involves a large, in fact in general infinitely large, family $\mathcal{F}$ of learning tasks $C$. Learning is carried out simultaneously in two loops. The *inner loop* learning involves the learning of a single task $C$ by a neural network $\mathcal{N}$, in our case an LSNN. This inner loop learning process is intended to be biologically realistic in our work. Some parameters of $\mathcal{N}$ (termed hyper-parameters) are optimized in an *outer loop* optimization to support fast learning of a randomly drawn task $C$ from $\mathcal{F}$. The outer loop training – implemented here through BPTT – proceeds on a much larger time scale than the inner loop, integrating performance evaluations from many different tasks $C$ of the family $\mathcal{F}$. One can interpret it in a biological setting as a process that mimics the impact of evolutionary and developmental optimization processes, as well as prior learning, on a functional level. Like in [9] we let all synaptic weights of $\mathcal{N}$ belong to the set of hyper-parameters that are optimized through the outer loop, i.e., through meta-learning. Hence the network is forced to encode all results from learning the current task $C$ in its internal state, in particular in its short-term memory. Thus the synaptic weights of the neural network $\mathcal{N}$ encode an efficient *algorithm* for learning arbitrary tasks $C$ from $\mathcal{F}$, rather than results of learning a particular task $C$.

### 3 Results on computational capabilities of LSNNs

**Sequential MNIST:** We tested the performance of LSNNs on a standard benchmark task that requires continuous updates of short term memory over a long time span: sequential MNIST [28][29]. The task is to classify the handwritten digits of the MNIST datasets when the pixels are presented sequentially, one after the other, row after row, see Fig. [1A]. We compare the performance of LSNNs with that of LSTM networks. Size of the LSNN, in the case of full connectivity, was chosen to match the number of parameters of the LSTM network. This led to 140 regular spiking and 100 adaptive neurons in comparison to 128 LSTM units. Actually it turned out that the sparsely connected LSNN shown in Fig. [1C], which had only 12% of the synaptic connections of the competing LSTM network, performed significantly better.

The 784 pixels are presented sequentially in 784 steps, after which the network is required to output the class of the presented digit. The grey values of pixels were given directly to artificial neural networks (ANNs), and encoded by spikes for RSNNs (one pixel per 1 ms). The top row of Fig. [1D] shows a version where the gray value of the currently presented pixel is encoded by population coding through the firing probability of the 80 input neurons. Somewhat better performance was achieved when each of the 80 input neurons is associated with a particular threshold for the gray value, and this input neuron fires whenever the gray value crosses its threshold in the transition from the previous to the current pixel (this input convention is chosen for the LSNN results of Fig. [1B]). In either case, an additional input neuron becomes active when the presentation of the 784 pixel values is finished, in order to prompt an output from the network. The firing of this additional input neuron is shown at the top right of the top panel of Fig. [1D]. The softmax of 10 linear output neurons $Y$ is trained through BPTT to produce, during this time segment, the label of the sequentially presented
The task is to classify images of handwritten digits when the pixels are shown sequentially pixel by pixel, in a fixed order row by row. B The performance of RSNNs is tested for two different setups: without adapting neurons (LIF), and an LSNN with randomly initialized connectivity that was rewired during training (LSNN). For comparison, the performance of two ANNs, an artificial fully connected RNN and an LSTM are also shown. C Connectivity (in terms of connection probabilities between and within the 3 subpopulations) of the LSNN after training. The input population X consists of 60 excitatory and 20 inhibitory neurons. Black (red) percentages on the arrows from X indicate connection probabilities from the excitatory (inhibitory) neurons. D Dynamics of the LSNN after training when the input image from A is sequentially presented. From top to bottom: spike rasters from input neurons (X), excitatory (E) and inhibitory (I) regularly spiking neurons, and adaptive neurons (A), only 50 neurons are shown; dynamics of the adaptive thresholds of a few adaptive neurons; activation of softmax readout neurons. A performance comparison is given in Fig. 1B. LSNNs achieve a 96% classification accuracy on the test set, the same as the LSTM network. The first and third bar in Fig. 1B show that this accuracy is out of reach for BPTT applied to spiking or nonspiking neural networks with the same number of weights but without the enhanced short term memory capabilities.

Speech recognition (TIMIT): We also tested the performance of LSNNs for a real-world speech recognition task, the TIMIT dataset. A thorough study of the performance of many variations of LSTM networks on TIMIT has recently been carried out in [30]. We used exactly the same setup of the TIMIT task which was used there (framewise classification) in order to facilitate comparison. We found that a standard LSNN consisting of 300 regularly firing (200 excitatory and 100 inhibitory) and 100 excitatory adapting neurons, with 20% connection probability in the network, achieved a classification error of 33.2%. This error is below the mean error around 40% from 200 trials with different hyperparameters for the best performing (and most complex) version of LSTMs according to Fig. 3 of [30], but above the mean of 29.7% of the 20 best performing choices of hyperparameters for these LSTMs. The performance of the LSNN turned out to be somewhat better than the error.
rates achieved in [30] for a less complex version of LSTMs without forget gates (mean of the best 20 trials: 34.2%).

We could not perform a similarly rigorous search over LSNN architectures and meta-parameters as was carried out in [30] for LSTMs. We tested however the variation where all adapting neurons were replaced by regularly firing excitatory neurons. It had a substantially higher error rate than the LSNN with adapting neurons: 37%. Details are given in the supplement.

4 Applying Learning-to-Learn to LSNNs

When the brain learns to predict sensory inputs, or the consequence of an action, this is a special case of learning from a teacher (i.e., supervised learning). The teacher is in this case the environment, which provides – often with some delay – the target value for a prediction. A key question is, which learning algorithm enables networks of neurons in the brain to carry out such learning from a teacher. The L2L results of [9] show that LSTM networks can learn nonlinear functions from a teacher without modifying their weights through BP or BPTT, using their short-term memory instead. We asked whether this interesting option for supervised learning could also be carried out by LSNNs.

We considered whether such alternative methods for supervised learning could also be demonstrated for RSNN. We considered a LSNN \( \mathcal{N} \) consisting 180 regularly firing neurons (population R) and 120 adapting neurons (population A) with full connectivity, rewiring did not improve performance in this case. All these neurons received input from a population X of 300 external input neurons. A linear readout received inputs from all neurons in \( \mathcal{R} \) and \( \mathcal{A} \).

We chose as family \( \mathcal{F} \) of tasks the class of all continuous functions of two real-valued variables \((x_1, x_2)\) that can be computed by a 2-layer artificial neural network of sigmoidal neurons with 10 neurons in the hidden layer, and weights and biases from [-1, 1], see Fig. 2A. Thus altogether, each such target network (TN) from \( \mathcal{F} \) was defined through 40 parameters from [-1, 1]: 30 weights and 10 biases. We give the teacher input for learning a particular TN \( C \) from \( \mathcal{F} \) like in [9] in a delayed manner: The target output value is given after \( \mathcal{N} \) has provided its guessed output value for the preceding input. This avoids that \( \mathcal{N} \) learns to cheat by simply using the teacher input as its output. The LSNN received a stream of 3 types of external inputs (see top row of Fig. 2C): the values of \( x_1, x_2 \), and of the output \( C(x'_1, x'_2) \) of the TN for the preceding input pair \( x'_1, x'_2 \) (set to 0 at the first trial), all represented through population coding in an external population of 100 spiking neurons. It produced outputs in the form of weighted spike counts during 20 ms windows from all neurons in the network (see bottom row of Fig. 2C), where the weights for this linear readout were trained, like all weights inside the LSNN, in the outer loop, and remained fixed during learning of a particular TN.

The training procedure in the outer loop of L2L was as follows: Network training was divided into training episodes. At the start of each training episode, a new target network TN was randomly chosen and used to generate target values \( C(x_1, x_2) \in [0, 1] \) for randomly chosen input pairs \((x_1, x_2)\). 500 of these input pairs and targets were used as training data, and presented one per step to the LSNN during the episode, where each step lasted 20ms. LSNN parameters were updated using BPTT to minimize the mean squared error between the LSNN output and the target in the training set, using gradients computed over batches of 10 such episodes, which formed one iteration of the outer loop. In other words, each weight update included gradients calculated on the input/target pairs from 10 different TNs. This training procedure forced the LSNN to adapt its parameters in a way that supported learning of many different TNs, rather than specializing on predicting the output of single TN. After training, the weights of the LSNN remained fixed, and it was required to learn the input/output behavior of TNs from \( \mathcal{F} \) that it had never seen before in an online manner by just using its short-term memory. See the suppl. for further details.

Most of the functions that are computed by TNs from the class \( \mathcal{F} \) are nonlinear, as illustrated in Fig. 2F for the case of inputs \((x_1, x_2)\) with \( x_1 = x_2 \). Hence learning the input/output behaviour of any such TN with biologically realistic local plasticity mechanisms presents a daunting challenge for a SNN. Fig. 2B shows that after a few thousand training iterations in the outer loop, the LSNN achieves low MSE for learning new TNs from the family \( \mathcal{F} \), significantly surpassing the performance of an optimal linear approximator (linear regression) that was trained on all 500 pairs of inputs and target outputs, see orange curve in Fig. 2B,D. In view of the fact that each TN is defined by 40
Figure 2: **LSNNs learn to learn from a teacher.**  

**A** Architecture of the two-layer feed-forward target networks (TNs) used to generate nonlinear functions for the LSNN to learn. **B** Performance of the LSNN in learning a new TN before (left) and after (right) training in the outer loop of L2L. **C** Network input (top row, only 100 of 300 neurons are shown), internal spike-based processing in the populations R and A (middle rows), and network output (bottom row) for 25 trials of 20ms each. **D** Learning performance of the LSNN for 10 new TNs. Performance for a single TN is shown as insert, a red cross marks step 7 after which output predictions became very good for this TN. The spike raster for this learning process is the one depicted in **C**. **E** Learning performance of BP for the same 10 TNs as in **D**, working directly on the ANN from **A**, with a prior for small weights. Its learning progress is slower compared with that of the LSNN shown in **D**. **F** Sample input/output curves of TNs on a 1D subset of the 2D input space, for different weight and bias values. These curves are all fairly smooth, like the internal models produced by the LSNN while learning a particular TN (panel **G**). **H** Illustration of the prior knowledge used by the LSNN for another family \( f \) (sinus functions). Even adversarially chosen examples (Step 4) do not induce the LSNN to forget its prior.
parameters, it comes at some surprise that the resulting network learning algorithm of the LSNN for learning the input/output behavior of a new TN produces in general a good approximation of the TN after just 5 to 20 trials (= randomly drawn labeled examples). The protocol of a generic learning process is shown in Fig. 2C. Each sequence of examples evokes an internal model that is stored in the short-term memory of the LSNN. Fig. 2D shows the fast evolution of internal models of the LSNN for the TN during the first trials (visualized for a 1D subset of the 2D input space). We make the current internal model of the LSNN visible by probing its prediction \( C(x_1, x_2) \) for hypothetical new inputs for evenly spaced points \((x_1, x_2)\) in the domain (without allowing it to modify its short-term memory; all other inputs advance the network state according to the dynamics of the LSNN). One sees that the internal model of the LSNN is from the beginning a smooth function, of the same type as the ones defined by the TNs in \( \mathcal{F} \). Within a few trials this smooth function approximated the TN quite well. Hence the LSNN had acquired during the training in the outer loop of L2L a prior for the types of functions that are to be learnt, that was encoded in its synaptic weights. This prior was in fact quite efficient, since Fig. 2D and E show that the LSNN was able to learn a TN with substantially fewer trials than a generic learning algorithm for learning the TN directly in an artificial neural network as in Fig. 2A: BP with a prior that favored small weights and biases (see end of Sec. 3 in suppl.). These results suggest that L2L is able to install some form of prior knowledge about the task in the LSNN. We conjectured that the LSNN fits internal models for smooth functions to the examples it received.

We tested this conjecture in a second, much simpler, L2L scenario. Here the family \( \mathcal{F} \) consisted of all sinus functions with arbitrary phase and amplitudes between 0.1 and 5. Fig. 2H shows that the LSNN also acquired in this setup from training in the outer loop an internal model for sinus functions (made visible analogously as in Fig. 2G). Even when we selected examples in an adversarial manner, which happened to be in a straight line, this did not disturb the prior knowledge of the LSNN.

Altogether the network learning that was induced through L2L in the LSNNs is of particular interest from the perspective of the design of learning algorithms, since we are not aware of previously documented methods for installing structural priors for online learning of a recurrent network of spiking neurons.

5 Applying meta-reinforcement learning to LSNNs

We now turn to an application of meta-RL to LSNNs. In meta-RL, the LSNN receives rewards instead of teacher inputs. Meta-RL has lead to a number of remarkable results for LSTM networks, see e.g. [6, 7]. In addition, [8] demonstrates that meta-RL provides a very interesting perspective of reward-based learning in the brain. We focused on one of the more challenging demos of [6] and [7], where an agent had to learn to find a target in a 2D maze, and to navigate subsequently to this target from random positions in the maze. Since LSNNs with just a few 100 neurons are not able to process visual input, we provided the current position of the agent within the maze through a place-cell like Gaussian population rate encoding of the current position.
An LSNN-based agent was trained on a family of navigation tasks in a circular arena during a sequence of episodes, each lasting 2 seconds. The goal was placed randomly for each episode on the border of the arena. When the agent reached the goal, it received a reward of 1, and was placed back randomly in the arena. The objective was to maximize the number of goals reached within the episode. This family \( \mathcal{F} \) of tasks is defined by the infinite set of possible goal positions. For each episode, an optimal agent is expected to explore until it finds the goal position, memorize it and exploits this knowledge until the end of the episode by taking the shortest path to the goal. We trained an LSNN so that the network could control the agent behaviour in all tasks, without changing its network weights.

For training in the outer loop, we used BPTT applied to the surrogate objective of the Proximal Policy Optimization (PPO) algorithm [31]. In this task the LSNN had 400 recurrent units (200 excitatory, 80 inhibitory and 120 adaptive neurons), the network was rewired with a fixed connectivity of 20%. The resulting network diagram and spike raster is shown in Suppl. Fig. 1. It received information about rewards in the form of spike inputs, like circuits in the PFC according to [8]. Altogether this demo shows that meta-RL can be applied to RSNNs, and produces previously not seen capabilities of sparsely firing RSNNs for complex control.

6 Discussion

We have demonstrated a new application of Deep Learning in computational neuroscience: It allows us to create architectures and learning algorithms for recurrent networks of spiking neurons (RSNNs) that provide superior computing and learning capabilities, and hence are more similar to RSNNs in the brain from the functional perspective. For that purpose we adapted BPTT so that it works efficiently for RSNNs, and can be combined with a biologically inspired synaptic rewiring method. We have shown in section 3 that this approach allows us create sparsely connected RSNNs that approach the performance of LSTM networks on common benchmark tasks for the classification of spatio-temporal patterns (sequential MNIST and TIMIT). This qualitative jump in the computational power of known RSNN architectures was supported by the introduction of adapting neurons into the model. These introduce a spread of longer time constants into RSNNs, as they do in the neocortex according to [20]. We refer to the resulting variation of the RSNN model as LSNNs, because of their longer short-term memory capability. This form of short-term memory is of particular interest from the perspective of energy efficiency of SNNs, because it stores and transmits stored information through non-firing of neurons.

We have shown in sections 4 and 5 that also learning-to-learn (L2L) and meta-reinforcement learning (meta-RL) can be ported in this way into RSNNs. In particular we have shown in Fig. 2 that one arrives in this way at new forms of network learning in RSNNs, that engage their short-term memory, rather than synaptic plasticity, for fast learning. It was recently argued in [8] that this form of fast learning is implemented in the PFC, and optimized through synaptic plasticity on a longer time scale. We have demonstrated in Fig. 2 that RSNN can learn in this way nonlinear input/output mappings that are usually thought to require an implementation of BP in the RSNN. Furthermore we have shown that this new form of network learning enables RSNNs, after suitable training with similar learning tasks in the outer loop of L2L, to learn a new task from the same class substantially faster. The reason is that the preceding training has installed abstract knowledge (priors) about common properties of these learning tasks in the RSNN, so that it does not have to learn a new task from scratch. To the best of our knowledge, transfer learning capabilities and the use of prior knowledge have previously not been demonstrated for SNNs. Fig 3 shows that meta-RL can be applied very successfully to RSNNs. It enables the RSNN – without any additional outer control or clock – to embody an agent that first searches a maze for a goal, and subsequently exploits the learnt knowledge in order to navigate fast from random initial positions to this goal.

Altogether we expect that the new methods for the investigation of computing and learning capabilities of RSNNs that we have introduced will have substantial benefits for research on understanding and reverse engineering of RSNNs in the brain. For example, the RSNNs that emerged in Fig. 1 and 3 all compute and learn with a brain-like sparse firing activity, quite different from a SNN that operates with rate-codes. In addition, these RSNNs present new functional uses of short-term memory that go far beyond remembering a preceding sensory input. Apart from these implications for computational neuroscience, our finding that RSNNs can acquire powerful computing and learning
capabilities with very energy-efficient sparse firing activity provides new application paradigms for spike-based computing hardware.

References


We provide in this supplement detailed information on the models and simulations of the main text, structured according to the corresponding sections therein.

2 LSNN model and deep learning methods

Neuron model: In continuous time the spike trains \( x_i(t) \) and \( z_j(t) \) are formalized as sums of Dirac pulses. Neurons are modeled according to a standard adaptive leaky integrate-and-fire model. A neuron \( j \) spikes as soon as its membrane potential \( V_j(t) \) is above its threshold \( B_j(t) \). At each spike time \( t \), the membrane potential \( V_j(t) \) is reset by subtracting the current threshold value \( B_j(t) \) and the neuron enters a strict refractory period where it cannot spike again. Importantly at each spike the threshold \( B_j(t) \) of an adaptive neuron is increased by a constant \( \beta/\tau_{a,j} \). Then the threshold decays back to a baseline value \( b^0_j \). Between spikes the membrane voltage \( V_j(t) \) and the threshold \( B_j(t) \) are following the dynamics

\[
\tau_m \dot{V}_j(t) = -V_j(t) + R_m I_j(t) \qquad (3)
\]
\[
\tau_{a,j} \dot{B}_j(t) = b^0_j - B_j(t), \qquad (4)
\]

where \( \tau_m \) is the membrane time constant, \( \tau_{a,j} \) is the adaptation time constant and \( R_m \) is the membrane resistance. The input current \( I_j(t) \) is defined as the weighted sum of spikes from external inputs and other neurons in the network:

\[
I_j(t) = \sum_i W_{ji}^{in} x_i(t - d_{ji}^{in}) + \sum_i W_{ji}^{rec} z_i(t - d_{ji}^{rec}), \qquad (5)
\]

where \( W_{ji}^{in} \) and \( W_{ji}^{rec} \) denote respectively the input and the recurrent synaptic weights and \( d_{ji}^{in} \) and \( d_{ji}^{rec} \) the corresponding synaptic delays. All network neurons are connected to a population of readout neurons with weights \( W_{kj}^{out} \). When network neuron \( j \) spikes, the output synaptic strength \( W_{kj}^{out} \) is added to the membrane voltage \( y_k(t) \) of all readout neurons \( k \). \( y_k(t) \) also follows the dynamics of a leaky integrator \( \tau_m \dot{y}_k(t) = -y_k(t) \).

Implementation in discrete time: Our simulations were performed in discrete time with a time step \( \delta t = 1 \) ms. In discrete time, the spike trains are modeled as binary sequences \( x_i(t), z_j(t) \in \{0, \frac{1}{\delta t}\} \), so that they converge to sums of Dirac pulses in the limit of small time steps. Neuron \( j \) emits a spike at time \( t \) if it is currently not in a refractory period, and its membrane potential \( V_j(t) \) is above its threshold \( B_j(t) \). During the refractory period following a spike, \( z_j(t) \) is fixed to 0. The dynamics of the threshold is defined by \( B_j(t) = b^0_j + \beta z_j(t) \) where \( \beta \) is a constant which scales the deviation \( b_j(t) \) from the baseline \( b^0_j \). The neural dynamics in discrete time reads as follows

\[
V_j(t + \delta t) = \alpha V_j(t) + (1 - \alpha) R_m I_j(t) - B_j(t) z_j(t) \delta t \qquad (6)
\]
\[
b_j(t + \delta t) = \rho_j b_j(t) + (1 - \rho_j) z_j(t), \qquad (7)
\]

where \( \alpha = \exp\left(-\frac{\delta t}{\tau_m}\right) \) and \( \rho_j = \exp\left(-\frac{\delta t}{\tau_{a,j}}\right) \). The term \( B_j(t) z_j(t) \delta t \) implements the reset of the membrane voltage after each spike. The current \( I_j(t) \) is the weighted sum of the incoming spikes. The definition of the input current in equation (5) holds also for discrete time, with the difference that spike trains now assume values in \( \{0, \frac{1}{\delta t}\} \).

Propagation of gradients in recurrent networks of LIF neurons: In artificial recurrent neural networks such as LSNNs, gradients can be computed with backpropagation through time (BPTT).
For BPTT in spiking neural networks, complications arise from the non-differentiability of the output of spiking neurons, and from the fact that gradients need to be propagated either through continuous time or through many time steps if time is discretized. Therefore, in [1][2] it was proposed to use a pseudo-derivative.

\[
\frac{dv_j(t)}{dv_j(t)} := \max\{0, 1 - |v_j(t)|\},
\]

where \(v_j(t)\) denotes the normalized membrane potential \(v_j(t) = \frac{V_j(t) - B_j(t)}{B_j(t)}\). This made it possible to train deep feed-forward networks of deterministic binary neurons [1][2]. We observed that this convention tends to be unstable for very deep (unrolled) recurrent networks of spiking neurons. To achieve stable performance we dampened the increase of back propagated errors through spikes by using a pseudo-derivative of amplitude \(\gamma < 1\) (typically \(\gamma = 0.3\)):

\[
\frac{dz_j(t)}{dv_j(t)} := \gamma \max\{0, 1 - |v_j(t)|\}.
\]

Note that in adaptive neurons, gradients can propagate through many time steps in the dynamic threshold. This propagation is not affected by the dampening.

**Rewiring and weight initialization of excitatory and inhibitory neurons:** In all experiments except those reported in Fig. 2, the neurons were either excitatory or inhibitory. When the neuron sign were not constrained, the initial network weights were drawn from a Gaussian distribution \(W_{ji} \sim \frac{w_0}{\sqrt{n_{in}}} \mathcal{N}(0, 1)\), where \(n_{in}\) is the number of afferent neurons in the considered weight matrix (i.e., the number of columns of the matrix), \(\mathcal{N}(0, 1)\) is the zero-mean unit-variance Gaussian distribution and \(w_0\) is a weight scaling factor chosen to be \(w_0 = \frac{100}{R_m} \delta t\). With this choice of \(w_0\) the resistance \(R_m\) becomes obsolete but the vanishing-exploding gradient theory [32][33] can be used to avoid tuning by hand the scaling of \(W_{ji}\). In particular the scaling \(\frac{1}{\sqrt{n_{in}}}\) used above was sufficient to initialize networks with realistic firing rates and that can be trained efficiently.

When the neuron sign were constrained, all outgoing weights \(W^{exc}_{ji}\) or \(W^{out}_{ji}\) of a neuron \(i\) had the same sign. In those cases, DEEP R [4] was used as it maintains the sign of each synapse during training. The sign is thus inherited from the initialization of the network weights. This raises the need of an efficient initialization of weight matrices for given fractions of inhibitory and excitatory neurons. To do so, a sign \(\kappa_i \in \{-1, 1\}\) is generated randomly for each neuron \(i\) by sampling from a Bernoulli distribution. The weight matrix entries are then sampled from \(W_{ji} \sim \kappa_i \mathcal{N}(0, 1)\) and post-processed to avoid exploding gradients. Firstly, a constant is added to each weight so that the sum of excitatory and inhibitory weights onto each neuron \(j\) (\(\sum_i W_{ji}\)) is zero [34] (if \(j\) has no inhibitory or no excitatory incoming connections this step is omitted). To avoid exploding gradients it is important to scale the weight so that the largest eigenvalue is lower of equal to 1 [32]. Thus, we divided \(W_{ji}\) by the absolute value of its largest eigenvalue. When the matrix is not square, eigenvalues are ill-defined. Therefore, we first generated a large enough square matrix and selected the required number of rows or columns with uniform probabilities. The final weight matrix is scaled by \(w_0\) for the same reasons as before.

To initialize matrices with a sparse connectivity, dense matrices were generated as described above and multiplied with a binary mask. The binary mask was generated by sampling uniformly the neuron coordinates that were non-zero at initialization. DEEP R maintains the initial connectivity level throughout training by dynamically disconnecting synapses and reconnecting others elsewhere. The \(L_1\)-norm regularization parameter of DEEP R was set to 0.01 and the temperature parameter of DEEP R was left at 0.

### 3 Results on computational capabilities of LSNNs

**MNIST setup:** The pixels of an MNIST image were presented sequentially to the LSNN in 784 time steps. Two input encodings were considered. First, we used a population coding where the gray scale value (which is in the range \([0, 1]\)) of the currently presented pixel was directly used as the firing probability of each of the 80 input neurons in that time step.
In a second type of input encoding – that is closer to the way how spiking vision sensors encode their input – each of the 80 input neurons was associated with a particular threshold for the gray value, and this input neuron fired whenever the gray value of the currently presented pixel crossed its threshold. Here, we used two input neurons per threshold, one spiked at threshold crossings from below, and one at the crossings from above. This input convention was chosen for the LSNN results of Fig. 1B.

The output of the network was determined by averaging the readout output over the 56 ms following the presentation of the digit. The network was trained by minimizing the cross entropy error between the softmax of the averaged readout and the label distributions. Rewiring with a global connectivity level of 12% was used during training to optimize a sparse network connectivity structure (i.e., when randomly picking two neurons in the network, the probability that they would be connected is 0.12). This implies that only a fraction of the parameters were finally used as compared to a similarly performing LSTM network.

**TIMIT setup:** To investigate if the performance of LSNNs can scale to real world problems, we attempted to solve the TIMIT speech recognition task. We focused on the frame-wise classification where the LSNN has to classify each audio-frame to one of the 61 phoneme classes.

We followed the convention of Halberstadt [35] for grouping of training, validation, and testing sets (3696, 400, and 192 sequences respectively). The performance was evaluated on the **core test set** for consistency with the literature. Raw audio is preprocessed into 13 Mel Frequency Cepstral Coefficients (MFCCs) with frame size 10 ms and on input window of 25 ms. We computed the first and the second order derivatives of MFCCs and combined them, resulting in 39 input channels. These 39 input channels were mapped to 39 input neurons which unlike in MNIST emit continuous values \( x_i(t) \) instead of spikes, and these values were directly used in equation 5 for the currents of the postsynaptic neurons.

Since we simulated the LSNN network in 1 ms time steps, every input frame which represents 10 ms of the input audio signal was fed to the LSNN network for 10 consecutive 1 ms steps. The softmax output of the LSNN was averaged over every 10 steps to produce the prediction of the phone in the current input frame. The LSNN was rewired with global connectivity level of 20%.

**Parameter values:** For adaptive neurons, we used \( \beta_j = 1.8 \), and for regular spiking neurons we used \( \beta_j = 0 \) (i.e. \( B_j \) is constant). The baseline threshold voltage was \( b_j^0 = 0.01 \) and the membrane time constant \( \tau_m = 200 \) ms. Networks were trained using the default Adam optimizer, and a learning rate initialized at 0.01. The dampening factor for training was \( \gamma = 0.3 \).

For sequential MNIST, all networks were trained for 36,000 iterations with approximately 66,000 parameters, with a batch size of 256. Learning rate was decayed by a factor 0.8 every 2500 iterations. The adaptive neurons in the LSNN had an adaptation time constant \( \tau_a = 700 \) ms. The baseline artificial RNN contained 128 hidden units with the hyperbolic tangent activation function. The LIF network was formed by a fully connected population of 220 regular spiking neurons.

For TIMIT, the LSNN network consisted of 300 regular neurons and 100 adaptive neurons which resulted in approximately 400,000 parameters. Network was trained for 80 epochs with batches of 32 sequences. Adaptation time constant of adaptive neurons was set to \( \tau_a = 200 \) ms. Refractory period of the neurons was set to 2 ms, the membrane time constant of the output Y neurons to 3 ms, and the synaptic delay was randomly picked from \{1, 2\} ms.

We note that due to the rewiring of the LSNN using DEEP R [4] method, only a small fraction of the weights had non-zero values (8,000 in MNIST, 80,000 in TIMIT).

4 Applying Learning-to-Learn to LSNNs

**Experimental setup:**

**Function families:** The LSNN was trained to implement a regression algorithm on a family of functions \( \mathcal{F} \). Two specific families were considered: In the first function family, the functions were defined by feed-forward neural networks with 2 inputs, 1 hidden layer consisting of 10 hidden neurons, and 1 output, where all the parameters (weights and biases) were chosen uniformly randomly.
between $[-1, 1]$. The inputs were between $[-1, 1]$ and the outputs were scaled to be between $[0, 1]$. We call these networks Target Networks (TNs). In the second function family, the targets were defined by sinusoidal functions $y = A \sin(\phi + x)$ over the domain $x \in [-5, 5]$. The specific function to be learned was defined then by the phase $\phi$ and the amplitude $A$, which were chosen uniformly random between $[0, \pi]$ and $[0.1, 5]$ respectively.

**Input encoding:** Analog values were transformed into spiking trains to serve as inputs to the LSNN as follows: For each input component, 100 input neurons are assigned values $m_{1}, \ldots, m_{100}$ evenly distributed between the minimum and maximum possible value of the input. Each input neuron has a Gaussian response field with a particular mean and standard deviation, where the means are uniformly distributed between the minimum and maximum values to be encoded, and with a constant standard deviation. More precisely, the firing rate $r_i$ (in Hz) of each input neuron $i$ is given by $r_i = r_{max} \exp \left( -\frac{(m_i - z_i)^2}{2\sigma^2} \right)$, where $r_{max} = 200$ Hz, $m_i$ is the value assigned to that neuron, $z_i$ is the analog value to be encoded, and $\sigma = \frac{(m_{max} - m_{min})}{1000}$, $m_{min}$ with $m_{max}$ being the minimum and maximum values to be encoded.

**LSNN setup and training schedule:** The standard LSNN model was used, with 300 hidden neurons for the TN family of learning tasks, and 100 for the sinusoidal family. Of these, 40% were adaptive in all simulations. There were recurrent connections within the regular and adaptive populations, and there were interconnections between regular and adaptive neurons, i.e., we used all-to-all connectivity between all neurons (regular and adaptive). The output of the LSNN was a linear readout that received as input the mean firing rate of each of the neurons per step i.e. the number of spikes divided by 20 for the 20 ms time window that the step consists of.

The network training proceeded as follows: A new target function was randomly chosen for each **episode** of training, i.e., the parameters of the target function are chosen uniformly randomly from within the ranges above (depending on whether its a TN or sinusoidal). Each **episode** consisted of a sequence of 500 **steps**, each lasting for 20 ms. In each step, one training example from the current function to be learned was presented to the LSNN. In such a step, the inputs to the LSNN consisted of a randomly chosen vector $x$ with its dimensionality $d$ and range determined by the target function being used ($d = 2$ for TNs, $d = 1$ for sinusoidal target function). In addition, at each step, the LSNN also got the target value $C(x')$ from the previous step, i.e., the value of the target calculated using the target function for the inputs given at the previous step (in the first step, $C(x')$ is set to 0).

All the weights of the LSNN were updated using our variant of BPTT, once per **iteration**, where an **iteration** consists of a batch of 10 **episodes**, and the weight updates are accumulated across episodes in an iteration. The Adam variant of BP was used with standard parameters and a learning rate of 0.001. The loss function for training was the mean squared error (MSE) of the LSNN predictions over an iteration (i.e. over all the steps in an episode, and over the entire batch of episodes in an iteration). In addition, a regularization term was used to maintain a firing rate of 20 Hz. Specifically, the regularization term $R$ is defined as the mean squared difference between the average neuron firing rate in the LSNN and a target of 20 Hz. The total loss $L$ was then given by $L = MSE + 30 \cdot R$. In this way, we induce the LSNN to use sparse firing. We trained the LSNN for 5000 iterations in all cases.

**Parameter values:** The LSNN parameters were as follows: 5 ms neuronal refractory period, delays spread uniformly between $0 - 5$ ms, adaptation time constants of the adaptive neurons spread uniformly between $1 - 1000$ ms, $\beta = 1.6$ for adaptive neurons (0 for regular neurons), membrane time constant $\tau = 20$ ms, 0.03 mV baseline threshold voltage. The dampening factor for training was $\gamma = 0.4$.

**Analysis and comparison:** The linear baseline was calculated using linear regression with L2 regularization with a regularization factor of 100 (determined using grid search), using the mean spiking trace of all the neurons. The mean spiking trace was calculated as follows: First the neuron traces were calculated using an exponential kernel with 20 ms width and a time constant of 20 ms. Then, for every step, the mean value of this trace was calculated to obtain the mean spiking trace. In Fig. 2B, for each episode consisting of 500 steps, the mean spiking trace from a random subset of 450 steps was used to train the linear regressor, and the mean spiking trace from remaining 50 steps was used to calculate the test error. The reported baseline is the mean of the test error over one batch of 10 episodes with error bars of one standard deviation. In Fig. 2D, for each episode, after every
Figure S1: **Meta-RL results for an LSNN.** **A** Samples of paths after training. **B** Connectivity between sub-populations of the network after training. The global connectivity in the network was constrained to 20%. **C** The network dynamics that produced the behavior shown in A. Raster plots and thresholds are displayed as in Fig. 1.D, only 1 second and 100 neurons are shown in each raster plots.

For the case where neural networks defined the function family, the test MSE was 0.0056 ± 0.0039 (linear baseline MSE was 0.0217 ± 0.0046). For the sinusoidal function family, the test MSE was 0.3134 ± 0.2293 (linear baseline MSE was 1.4592 ± 1.2958).

**Comparison with backprop:** The comparison was done for the case where the LSNN is trained on the function family defined by neural networks. A feed-forward (FF) network with 10 hidden neurons and 1 output was constructed. The input to this FF network were the analog values that were used to generate the spiking input and targets for the LSNN. Therefore the FF had 2 inputs, one for each of $x_1$ and $x_2$. The error reported in Fig 2E is the mean training error over 10 batches with error bars of one standard deviation.

The FF network was initialized with Xavier normal initialization \(^{[36]}\) (which had the best performance, compared to Xavier uniform and plain uniform between $[-1, 1]$). Adam \(^{[37]}\) with AMSGrad \(^{[38]}\) was used with parameters $\eta = 10^{-1}$, $\beta_1 = 0.7$, $\beta_2 = 0.9$, $C = 10^{-5}$. These were the optimal parameters as determined by a grid search. Together with the Xavier normal initialization and the weight regularization parameter $C$, the training of the FF favoured small weights and biases.
5 Applying meta-reinforcement learning to LSNNs

Experimental setup:

Task family: An LSNN-based agent was trained on a family of navigation tasks in a two-dimensional circular arena. For all tasks, the arena is a circle with radius 1 and goals are smaller circles of radius 0.3 with centres uniformly distributed on the circle of radius 0.85. At the beginning of an episode and after the agent reaches a goal, the agent’s position is set randomly with uniform probability within the arena. At every timestep, the agent chooses an action by generating a small velocity vector of Euclidean norm smaller or equal to \( a_{\text{scale}} = 0.1 \). When the agent reaches the goal, it receives a reward of 1. If the agent attempts to move outside the arena, the new position is given by the intersection of the velocity vector with the border and the agent receives a negative reward of \(-0.05\).

Input encoding: Information of the current environmental state \( s(t) \) and the reward \( r(t) \) were provided to the LSNN at each time step \( t \) as follows: The state \( s(t) \) is given by the \( x \) and \( y \) coordinate of the agent’s position. Each position coordinate \( \xi_i \) is encoded by 40 neurons which spike according to a Gaussian population code. Each neuron is assigned to each of the 40 neurons, where \( \xi_i \)’s are evenly spaced between \(-1\) and \(1\). The firing rate of neuron \( i \) is then given by \( r_{\max} \exp(-100(\xi_i - \xi)^2) \) where \( r_{\max} \) is 500 Hz. The instantaneous reward \( r(t) \) is encoded by two groups of 40 neurons. All neuron in the first group spike in synchrony whenever a reward of 1 is received (i.e., the goal was reached), and the second group spikes when a reward of \(-0.05\) is received (i.e., the agent moved into a wall).

Output decoding: The output of the LSNN is provided by five readout neurons. Their membrane potentials \( y_i(t) \) define the outputs of the LSNN. The action vector \( a(t) = (a_x(t), a_y(t))^T \) is sampled from a distribution \( \pi_\theta \) which depends on the network parameters \( \theta \) through the readsouts \( y_i(t) \) as follows: The coordinate \( a_x(t) \) is sampled from a Gaussian distribution with mean \( \mu_x = \tanh(y_1(t)) \) and variance \( \sigma_x = \sigma(y_2(t)) \). The firing rate that updates the agent’s position is then defined as \( a_{\text{scale}} \cdot a(t) \). If this velocity has a norm larger than \( a_{\text{scale}} \), it is clipped to a norm of \( a_{\text{scale}} \).

The last readout output \( y_5(t) \) is used to predict the value function \( V_0(t) \). It estimates the expected discounted sum of future rewards \( R(t) = \sum_{t'=t}^{t'=T} \eta^{t'-t} r(t') \), where \( \eta = 0.9 \) is the discount factor and \( r(t') \) denotes the reward at time \( t' \). To enable the network to learn complex forms of exploration we introduced current noise in the neuron model in this task. At each time step, we added a small Gaussian noise with mean 0 and standard deviation \( R_m \nu_j \) to the current \( I_j \) into neuron \( j \). Here, \( \nu_j \) is a network parameter initialized at 0.03 and optimized by BPTT alongside the network weights.

Network training: To train the network we used the Proximal Policy Optimization algorithm (PPO) \cite{schulman2017proximal}. For each training iteration, \( K \) full episodes of \( T \) timesteps were generated with fixed parameters \( \theta_{\text{old}} \) (here \( K = 10 \) and \( T = 2000 \)). We write the clipped surrogate objective of PPO as \( O^{\text{PPO}}(\theta_{\text{old}}, \theta, t, k) \) (this is defined under the notation \( L^{\text{CLIP}} \) in \cite{schulman2017proximal}). The loss with respect to \( \theta \) is then defined as follows:

\[
\mathcal{L}(\theta) = -\frac{1}{KT} \sum_{k<K, t<T} O^{\text{PPO}}(\theta_{\text{old}}, \theta, t, k) + \mu_v (R(t, k) - V_0(t, k))^2
\]

\[
-\mu_v H(\pi_\theta(k, t)) + \mu_{\text{firing}} \frac{1}{N} \sum_{j=1}^{N} \frac{1}{KT} \sum_{k, t} ||z_j(t, k) - f^0||^2, \tag{11}
\]

where \( H(\pi_\theta) \) is the entropy of the distribution \( \pi_\theta \), \( f^0 \) is a target firing rate of \( 10 \) Hz, and \( \mu_v, \mu_v, \mu_{\text{firing}} \) are regularization hyper-parameters. Importantly probability distributions used in the definition of the loss \( \mathcal{L} \) (i.e. the trajectories) are conditioned on the current noises, so that for the same noise and infinitely small parameter change from \( \theta_{\text{old}} \) to \( \theta \) the trajectories and the spike trains are the same. At each iteration this loss function \( \mathcal{L} \) is then minimized with one step of the ADAM optimizer.

Parameter values: In this task the LSNN had 400 hidden units (200 excitatory neurons, 80 inhibitory neurons and 120 adaptive neurons) and the network was rewired with a fixed global connectivity of 20\% \cite{gerstner2014spiking}. The adaptation time constants were chosen randomly with uniform probability
within 200 ms and 4 s. The membrane time constants were similarly sampled between 15 and 30 ms. The adaptation amplitude $\beta$ was set to 1.7. The refractory period was set to 3 ms and delays were sampled uniformly between 1 and 10 ms. The regularization parameters $\mu_v$, $\mu_e$ and $\mu_{firing}$ were respectively 0.05, 1, and 100. The parameter $\epsilon$ of the PPO algorithm was set to 0.2. The learning rate was initialized to 0.01 and decayed by a factor 0.95 every 100 iterations. We used the default parameters for ADAM, except for the parameter $\epsilon$ which we set to $10^{-5}$.