Ceca Kraišniković, BSc

Symbolic Computation in Spiking Neural Networks

Master's Thesis

to achieve the university degree of

Diplom-Ingenieurin

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor:
Em.Univ.-Prof. Dipl.-Ing. Dr.rer.nat. Wolfgang Maass
Institute of Theoretical Computer Science

Graz, December 2017

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Scientific progress is made step by step. If we dream of great things, for example, to understand the complex functioning and mechanisms of our brain, and do *tiny* steps in that direction, one day we will be able to achieve what we seek for. Inspired by the prefrontal cortex, where abstract rule-like representations guide our behavior and other cognitive processes, we were motivated to explore learning capabilities of Spiking Neural Networks (SNNs) in the task Symbolic Computation. For symbolic or algebraic computation the term *symbolic (mathematical) expression* is important. We will focus on using simple arithmetic operations, namely, addition, subtraction and multiplication. Humans can, without many difficulties, use symbolic expressions describing how to perform simple computation of, say, two numbers. If we want to have (more) intelligent machines, they should be able to do this as well. So, our task now is to help them make progress and be able to perform simple symbolic computation as described by the abstract rule (symbolic expression).

According to Dan Brown, *"Everything is possible. The impossible just takes longer"* [1].

Zusammenfassung

Wissenschaftlicher Fortschritt wird Schritt für Schritt gemacht. Wenn wir von großen Dingen träumen, zum Beispiel, die komplexen Funktionen und Mechanismen unseres Gehirns zu verstehen, und auch nur winzige Schritte in diese Richtung machen, werden wir eines Tages erreichen, was wir anstreben. Inspiriert von dem präfrontalen Cortex, wo regelähnliche Darstellungen unser Verhalten und andere kognitive Prozesse leiten, waren wir motiviert, die Lernmöglichkeiten von Gepulsten Neuronalen Netzwerken in der Aufgabe Symbolischer Berechnung zu erforschen. Für symbolische oder algebraische Berechnung ist der Begriff *symbolischer (mathematischer) Ausdruck* wichtig. Wir werden den Schwerpunkt auf einfache arithmetische Operationen, und zwar Addition, Subtraktion und Multiplikation, legen. Menschen können ohne Schwierigkeiten symbolische Ausdrücke verwenden, die beschreiben, wie beispielsweise eine einfache Berechnung von zwei Zahlen durchgeführt wird. Wenn wir intelligente(re) Maschinen möchten, sollten diese das auch können. Deswegen ist es jetzt unsere Aufgabe ihnen zu helfen, weitezukommen, sodass sie eine einfache symbolische Berechnung, beschrieben durch eine abstrakte Regel (einen symbolischen Ausdruck), leisten können.

Wie Dan Brown schreibt: *"Alles ist möglich. Das Unmögliche dauert nur etwas länger"* [1].

Acknowledgment

For the great support and motivation, special thanks to:

my late and dear grandpa, wherever he is now,

my parents, sister and brother, always being patient and encouraging,

my supervisor Wolfgang Maass, for this outstanding opportunity to be part of his research team,

Anand Subramoney, for all kind of advices (ranging from Python to purpose of life and life),

all IGI family, our journal club sessions and the nights going out for a dinner, *Bier* and *Schnaps*,

Michael Müller, for providing me with this great template and

all other friends, particularly those who have their *home* address at Inffeld campus as well.

Contents

List of Figures

List of Tables

1 Introduction

Artificial intelligence (AI) is very active research field nowadays. The AI approach appeared with the digital computer, which can be specialized for different computations. We, humans, are intelligent beings and one important characteristic of our intelligence is the ability to manipulate abstract symbols. Computers do that too – they manipulate abstract symbols in order to perform computations. People started thinking about intelligence and how to create programs that do what we can do. Whether we use words and grammar rules or mental symbols which represent names or properties, we do it using brain's network of neurons. On the other side, a digital computer is a system composed of millions of logic gates, wired into circuits [2].

The very first step towards integrating two fields of research, namely, theoretical neurophysiology and the theory of propositional logic, has been made in 1943, by McCulloch and Pitts [3]. McCulloch and Pitts pointed out that neurons could be similar to logic gates, since they gather inputs from each other, process those inputs and depending on the processing mechanism, fire off an output or not. Neurons can, in theory, be seen as living logic gates, which implement digital functions [3], hence this was the first model resembling the functionality of a biological neuron.

After World War II, digital computers became available for broader applications. The pioneers of AI started with language translation and image and speech understanding applications, based on the previously mentioned concepts of McCulloch and Pitts model of a neuron. They claimed that computer intelligence would very quickly match and surpass human intelligence. In fact, successful applications were only good in solving the particular problems for which they were designed, but they could not generalize or be flexible in solving problems of similar nature [2]. Humans can do that without any difficulties, since we are able to transfer the knowledge we have acquired and hence, we can solve problems flexibly [4]. The challenge was (and still is) designing programs which would be able to solve problems which humans cannot describe easily by a list of formal, mathematical rules, but rather intuitively [5].

During years, available computational power made it possible for computers to be good in solving particular tasks and to compete with human level performance [4]. Nowadays, computers are able to learn complicated concepts by building them out of simpler ones and to gather knowledge from experience (for example, convolutional neural networks for pattern classification tasks). Many of the most advanced and intelligent solutions nowadays are outcome of the deep learning techniques, with neural network architectures designed for specific purposes. Computational units that become intelligent via their interactions with each other is the concept behind neural networks and it is inspired by the brain. In fact, neuroscience can give

us guidelines and ideas for underlying models of powerful applications [5], but despite showing great results and being biologically inspired, they are aimed for solving actual computational problems. Of course, this is valuable approach to understand our brain better, but successes of failures of such systems do not necessarily provide explanations to the question how our brain works [6]. In our quest to make a machine intelligent, we would need to understand much better the principles of brain and its computation.

Gary Marcus in his works [7, 8] writes about *a cognitive architecture*, trying to integrate two possible theories. Is the mind a manipulator of symbols or it is a large neural network with neuron-like nodes and synapse-like connections? Marcus also points out that deep learning networks have proved to be successful in the pattern classification tasks (for example, speech or image recognition), but have shown little progress in areas such as reasoning or natural language understanding. Circuits for encoding and manipulating sequences, manipulation and encoding of variables, working memory storage, decision-making and many other concepts are needed as computational primitives, alongside the primitive for hierarchical pattern recognition. He pinpoints that particularly important would be to understand the neural mechanisms of "variable binding" concept, which is of crucial importance in language and deductive reasoning, and possibly, fundamental for understanding the relations between neurons and higher-level cognitive processes. Variable binding term refers to connecting together two pieces of information - a variable and an arbitrary instance of that variable, for example, in algebra - abstract symbols X or Y with a number, in language - *subject* or *verb* with a word, in general - a placeholder with a symbol. Some of proposed neural mechanisms for variable binding are: binding through synchrony [9], vector symbolic architectures [10], precisely controlled recurrent interactions between the prefrontal cortex and basal ganglia [11].

On the other hand, Maass in his works writes that "neural networks are highly recurrent networks of neurons and synapses with diverse dynamic properties", that neurons in the brain code their information through the patterns (by spike trains) in which each neuron fires relative to other neurons in the group and that full spatial and temporal pattern is computationally relevant signal. Unlike the offline computation which requires a global synchronisation (like a PC, or a traditional artificial neural network, or in general, Turing machine), where all relevant inputs are available at the start of computation, and then processing (computation) gives the output, in online computation (performed by our brain as well) diverse pieces of information arrive at different points in time and the computation has to start before all of them are available. Additional pieces of information representing new inputs and results of previous subcomputations are integrated into computation while computation is performed, if necessary. Instead of global synchronisation to perform computations, time is used as a new dimension for coding information. These facts lead to creating a new generation of

neural networks - spiking neural networks [12, 13, 14, 15].

Hence, using biologically more realistic neural networks - spiking neural networks, and even improved model with adapting spiking neurons [16], binding together the previously mentioned and aspects proposed by Marcus, through this work we would like to demonstrate some interesting computational properties of recurrently connected networks of adapting spiking neurons, in the task where we use abstract rule-like representations for symbolic (algebraic) computation.

1.1 Motivation and Contributions

If we compare the number of neurons of our brain's network (about 10^{11}) with the number of transistors in a modern supercomputer, these numbers are in the same range, but unlike the efficient computing performed by our brain, supercomputers are great power consumers. In order to understand better energy-efficient computation performed by our brain and benefit from it, spike-based computations are of great interest to be studied. Spiking neural networks can be emulated in a more energy-efficient way using neuromorphic hardware, which is specially designed to mimic biological architectures [17]. For example, Intel's Loihi, fully digital and asynchronous neuromorphic chip, composed of neurons having local memory and state dynamics and communicating via spike impulses is on-chip trainable [18], and as such, a great target for new experiments with spiking neural networks. Some other popular neuromorphic chips are SpiNNaker [19], Brainscales [20], True North [21]. The benefits are twofold. On one hand, we aim to develop more powerful SNNs and, of course, the target could be neuromorphic hardware, on the other hand, examining capabilities and limitations of neuromorphic hardware using SNNs is of great significance.

Our aim is that we move, through this research, toward computation using "higher-level" rules. Although these "abstract rules" will be given and network would need to bind pieces of information, connecting two levels of abstraction (for example, a rule with concrete numbers), maybe this research could be the ground point for the further research where the network would be able to discover abstract rules by itself.

Our main task is to explore learning capabilities of LSTM-like units in a recurrently connected network of spiking neurons. First we will investigate whether the network is able to apply computational program only through identifier of that program. Later we switch to use rules for symbolic computation and, lastly, we test working memory of these LSTM-like units. All experiments will be performed in Learning-To-Learn (LTL) setup. Using LTL with spiking neural networks is also of great importance, because, so far, it is not studied enough.

2 Background

How do computers carry computations? Traditional computers are programmed in terms of instructions and variables. Unlike them, artificial neural networks are defined by a large set of numbers (parameters) which represent the strengths (weights) of synapses between neurons in the brain and the excitabilities (biases) of neurons. An optimization algorithm, used in an iterative process to adjust these parameters, is aimed at minimizing the errors for a concrete computational task. The underlying architectures and neural models are chosen in such a way to maximize the performance of particular learning algorithms for particular tasks, which is not necessarily similar to biological networks of neurons [22].

2.1 Perceptron and Sigmoidal Neuron Models

McCulloch-Pitts neuron, illustrated in Figure 1, also known as perceptron or threshold gate, represents the first model of a neuron in the first generation of artificial neural networks, designed to capture essential aspects of neural computation carried out in the brain. Conceptually it is very simple – a neuron outputs binary 'high' signal, if the sum of its weighted inputs surpasses a certain threshold value, otherwise the output is binary 'low' signal. More precisely, given weights $w_0, w_1, ..., w_n$ and inputs $x_1, ..., x_n$, the output y is computed as

$$x = \sum_{i=0}^{n} w_i x_i > 0, \tag{1}$$

$$y = f(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where each w_i represents a real-valued constant, also denoted as weight, that determines the contribution of input x_i to the output. The weight $(-w_0)$ is a threshold value that the weighted sum of elements $(w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n)$ must surpass in order for perceptron to output a 1. $f(x)$ denotes a step or threshold activation function. Hence, this model produces digital outputs (0 or 1), which leads to having many difficulties with designing learning algorithms for networks with several layers of such neurons [23, 24].

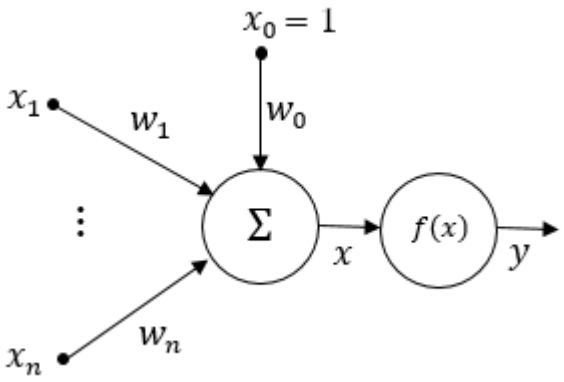

Figure 1: **A simple neuron model.** In case $f(x)$ is a step or threshold function, this model represents a perceptron (McCulloch-Pitts neuron). In case $f(x)$ is a sigmoidal function, this model represents a sigmoidal neuron with analog outputs between 0 and 1.

The second generation of artificial neural networks uses sigmoidal neuron model. This model uses a continuos activation function f, $f : \mathbb{R} \mapsto [0, 1]$, monotonically increasing and depicted in Figure 2, which interpolates in a smooth differentiable manner values between 0 and 1, making networks consisting of these neurons suitable for analog inputs and outputs. Neurons either fire or not, and an analog output can be interpreted as normalized firing rate (frequency) of the neuron within some time window. Hence, neurons of this generation are more biologically realistic than the neurons of the first generation.

Commonly used examples of networks with neurons of this type are feed-forward and recurrent neural networks. Learning algorithm used to train such networks is based on gradient-descent optimization. Differentiable activation function makes it possible to compute (using chain rule) how parameters of neurons contributing to network's output should be changed in order to reduce the error at the output. The chain rule is applied layer-wise, starting at the output layer, calculating and propagating errors back toward the input layer, hance, „backprop“ learning algorithm [22, 24, 13].

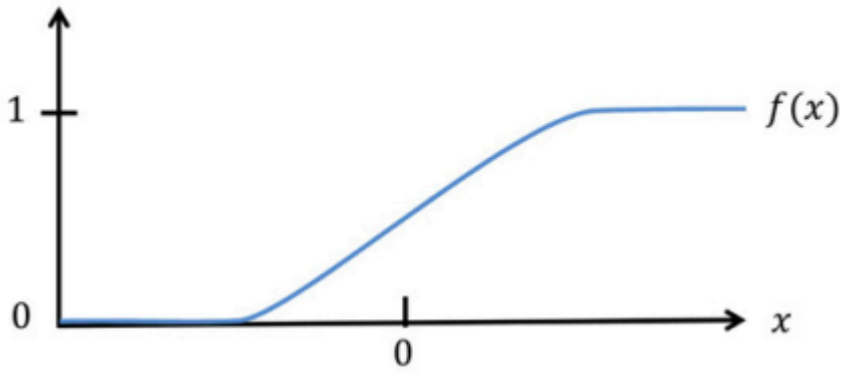

Figure 2: **A sigmoidal gate, a smooth activation function.** [From Maass, 2016]

2.2 Spiking Neural Networks

Computers and other devices for information processing consist of hardware and software. For their functioning, dry environment is required and pouring water over our computer will cause its damage – it will stop working. Here, one can see a sharp contrast between computers and with the absolute need for water all organisms in nature have. For living organisms, water is essential. An artificial sea-environment of salty extra-cellular fluid surrounds the neurons in our brain. Brain and parts of it (in short, our nervous system), are referred to as *wetware*. Similarities between the wetware in our brain and the wetware of the creatures still living in the sea made the research on neurons and brain in general easier, since, for example, squid have up to thousand times larger neurons than those in our brain. Despite the difference in size, the functioning mechanism is similar – the Hodgkin's and Huxley's equations for the dynamics of neuron apply both to squid and the neurons in our brain.

Neurons carry out computation, but it is essential that these intermediate results are communicated between neurons. So, in order for computation to be possible in wetware, nature had a challenging task to solve, namely, to find a way for neurons to communicate and use information via so-called spikes, also referred to as action potentials. While computers use bits sent over copper wires, neurons communicate using spikes. Similarily to a bit, a spike is the common unit of information in wetware.

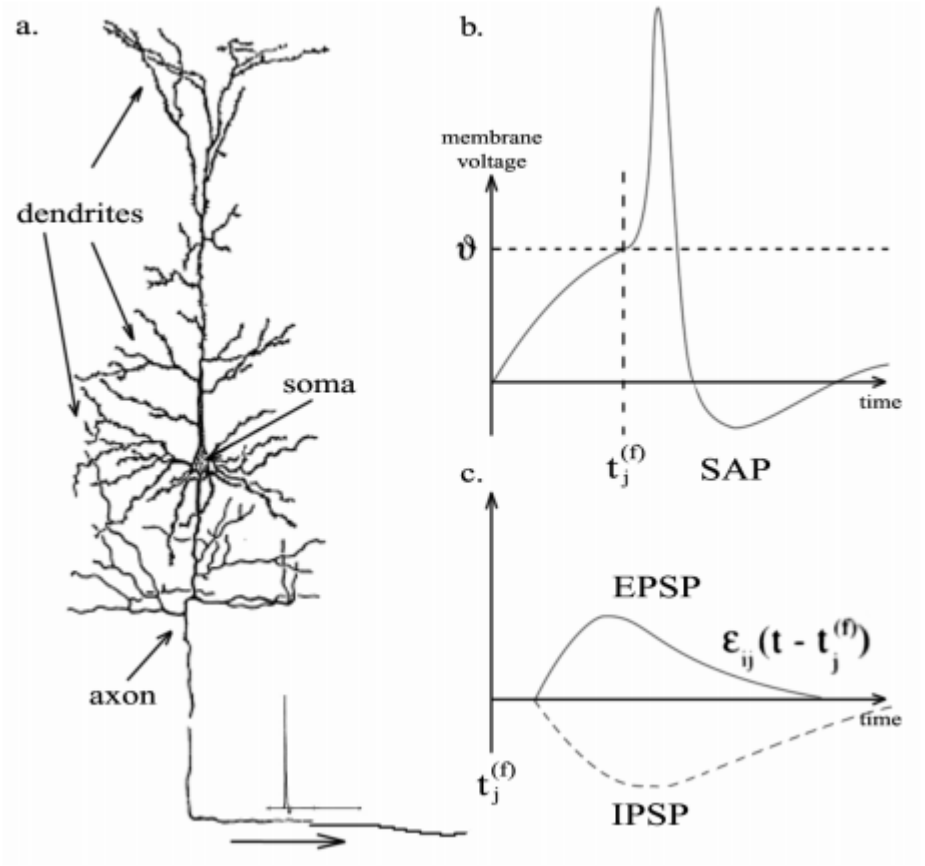

Figure 3: **A spiking neuron. a**. Simplified illustration of a neuron, with dendrites, soma and axons marked and a spike propagated along axon. **b**. Time course of one spike. As soon as membrane potential reaches a threshold ϑ at moment $t_j^{(f)}$, the neuron generates a spike and goes into a refractory state, here denoted as SAP (negative Spike After-Potential). **c**. Excitatory and Inhibitory Postsynaptic Potentials over time (EPSPs and IPSPs, respectively). SAP and postsynaptic potentials can be modeled using appropriate kernels. Here ε_{ij} denotes a kernel, which describes the effect of the postsynaptic potentials. [From Vreeken, 2003]

Essential parts of a neuron are cell body (soma), dendrites, axons and synapses.

Dendrites, also referred to as a „dendritic tree“, collect synaptic inputs. There are approximately 10000 inputs per neuron.

Cell body (soma) with axon hilloc represent action potential (spike) generator. A spike is a short ($1ms$) and a sudden increase in voltage, formed at the trigger zone of the soma and transmitted along *axons*, which grow quite long before they start to branch (analog to the axons in wetware, copper wires are signal carriers in hardware). The purpose of these branching points

is to duplicate the spikes, so that a single spike from one neurons can easily be transmitted to a few thousand other neurons, if necessary. Typically, axons transmit spike outputs to synapses in the dendritic trees of around 10000 other neurons.

Spikes move from one neuron to another only if they pass a very complex signal pre-processor, a so-called *synapse*, where a rather complicated chain of events at chemical level happens. As a result, an increase or decrese of the membrane voltage occurs. One refers to these voltage changes either as excitatory (EPSPs, excitatory postsynaptic potentials) or inhibitory (IPSPs, inhibitory postsynaptic potentials) potentials, as shown in Figure 3c.

EPSPs represent positive postsynaptic potential, hence, they increase the membrane voltage, whereas IPSPs represent negative postsynaptic potential, decreasing the membrane voltage. The resulting membrane voltage of a neuron is represented by a sum of many such continuosuly arriving EPSPs and IPSPs from other neurons, and once this sum reaches a firing threshold ϑ, the neuron fires, sending out a spike down the axon. In other words, neurons can excite or inhibit other neurons via synapses – they connect the axon of the presynaptic neuron to the dendrite (or soma) of the postsynaptic neuron.

Spikes are very much alike, but postsynaptic potentials differ in size and shape, depending on the properties of the synapses (synaptic efficacy) and the history of the synapse (e.g., „mood“ and recent „experiences“) [12, 24].

Neurons and synapses implement processors and memory for brain computations. But, how do neurons communicate? They communicate via spike trains, which are sequences of spikes. These sequences contain all the relevant information of the computation which should be transmitted from one neuron to another. One can take in account firing frequency of the neurons, but the changes in frequency are very rapid and the temporal distances between two spikes are very irregular, so, it is obvious that another aspects should be taken in account. Studies have shown that spatial and temporal dimension of information are relevant for encoding the messages neurons communicate to each other. The message is encoded through the spatio-temporal pattern – firing of each neuron relative to the firing of other neurons in the group [12].

Spiking neurons, as substantially more biologically realistic neuron model, are computational units utilized in spiking neural networks, i.e., the third generation of neural network models [13].

2.3 Leaky Integrate-and-Fire (LIF) Neurons

Leaky Integrate-and-Fire neuron model is one of the best-known models of spiking neurons. It is a thresholding model, more precisely, the spike generation mechanism is implemented by a threshold. Every time the membrane potential u crosses a threshold ϑ from below, spikes are generated. The

moment of spike being generated is referred to as a firing time, $t^{(f)}$,

$$t^{(f)} : u(t^{(f)}) = \vartheta. \tag{3}$$

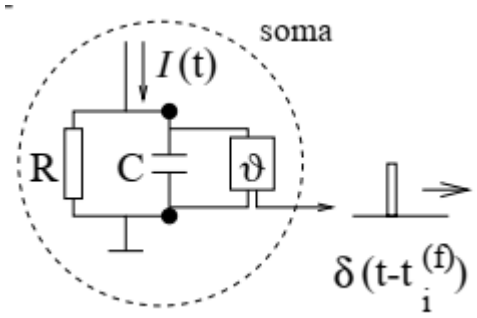

Figure 4: **A RC circuit.** An electrical equivalent to LIF neuron. [From Gerstner, 2002]

The basic circuit of integrate-and-fire model is shown in Figure 4. It consists of a resistor-capacitor circuit (RC circuit) driven by a current $I(t)$. For this circuit, we can write equations:

$$I(t) = I_R(t) + I_C(t), \tag{4}$$

$$I_R(t) = \frac{u(t)}{R}, \tag{5}$$

$$I_C(t) = C\frac{du}{dt}. \tag{6}$$

Expression (5) is calculated using Ohm's law, whereas (6) follows from the definition of the capacity, $C = q/u$ (then $q = uC$), and the definition of electric current, $I = dq/dt$, where q is the charge and u is the voltage. Thus,

$$I(t) = \frac{u(t)}{R} + C\frac{du}{dt}, \tag{7}$$

and transforming this expression, we get

$$\tau_m \frac{du}{dt} = -u(t) + RI(t), \tag{8}$$

where $\tau_m = RC$ represents the time constant of the "leaky integrator". The voltage u represents the membrane potential and the constant τ_m is the membrane time constant.

In general, after a spike is generated, the dynamics is reset and again according to (8) until the next spike occurs, but very often leaky integrate-and-fire model includes an absolute refractory period, during which the dynamics is interrupted and only reset after the refactory period, namely, at

the moment $t^{(f)} + \Delta^{abs}$, with the membrane potential u being set to the resting potential $u_r < \vartheta$ during the refractory period.

In fact, an external current $I(t)$ can be used to stimulate a neuron, but it is more realistic that neurons, since they are part of a larger network, are stimulated by a current resulting from the activity of presynaptic neurons. In this case, each presynaptic spike injects a current of some typical form $\alpha(s)$ into the membrane. Current-based neuron model can be then described as

$$I_i(t) = \sum_j w_{ij} \sum_f \alpha(t - t_j^{(f)}), \tag{9}$$

where each j denotes a presynaptic neuron relevant for the neuron i, w_{ij} represents synaptic efficacy from presynaptic neuron j to the neuron i and $t_j^{(f)}$ denotes f-th spike of presynaptic neuron j.

One commonly used mathematical model for α is the exponential model, with

$$\alpha(s) = \frac{q}{\tau_s} \exp(-\frac{s}{\tau_s})\Theta(s), \tag{10}$$

where q is the total charge injected in a postsynaptic neuron, τ_s is a time constant for decay and Θ is the Heaviside step function ($\Theta(s) = 1$ if $s > 0$, otherwise $\Theta(s) = 0$) [25].

2.4 Recurrent Neural Networks and LSNNs

In machine learning research, it has been shown that recurrent neural networks (RNNs) are computationally very powerful. They are very successfully applied in solving tasks which include sequential inputs (for example, speech and language or video recognition). This is because their architecture is suitable for processing one element of a sequence at each timestep and their hidden units are able to maintain the history of all elements presented earlier. In fact, they are able to integrate information over time, but essential to their remarkable successes is the use of special modules called Long-Short Term Memory (LSTM) units. Introduced by Hochreiter and Schmidhuber, in their paper Long Short-Term Memory (see [26]), these units are capable of learning long-term dependencies by using their store und update gates, hence, a basic LSTM unit represents a memory cell [27].

In contrast to great successes of recurrent networks with LSTM units, recurrent networks of spiking neurons (SNNs) are able to integrate information on time scale of hundreds of millisecond through short-time plasticity, which is not enough for tasks that require longer maintenance of relevant information. To overcome this problem and empower SNNs with greater capabilities, adapting spiking neurons are integrated in the network. These adapting neurons can be modelled by LIF neurons with an adaptive firing threshold and are referred to as ALIF neurons (Adaptive LIF). Analogously

to RNNs and LSTMs, ALIF neurons are LSTM-like units for SNNs, and hence we call such networks LSNNs [28, 16].

2.5 Learning-To-Learn (LTL)

As already mentioned, deep learning neural networks do perform very well in solving tasks they are specialized for and often they even outperform humans, but their learning does not include some of the cruical aspects of human learning. Human-like learning and thinking machines should be able to:

- explain and understand the world through causal thinking (causality),
- enrich and support the knowledge that is learned,
- use compositionality and learning-to-learn in order to rapidly acquire and generalize to new tasks and situations [4].

Here we will focus on the last aspect, particularly on Learning-To-Learn (LTL), since we use that approach in our experiments.

LTL is the approach which makes rapid model learning possible. Compositionality refers to the idea that combination of primitive elements leads to new representations. Given a finite set of primitives, an infinite number of new representations can be constructed. We are able to construct new models from previously learned parts and relations between them through learning-to-learn, indicating that these two concepts (compositionality and learning-to-learn) complement each other [4].

Having experience in doing something, finding two concepts similar or noticing that something is a special case of some more general concept indicate that the learner of a new task will learn faster or more accurately than the unexperienced learner. This comes as a consequence of solution regularities in a problem domain [29].

Meta-learning using gradient descent, introduced in [29], by Hochreiter, Younger and Conwell, is organized through two systems - supervisory and subordinate systems. Subordinate system is a recurrent neural network which is adjustable. A goal is to learn a target function f_k using sequences s_k from a set of sequences $\{s_k\}$, which are all generated using this target function f_k. Inputs to the meta-learning system are an example of the current function to learn and the result of the previous example.

Meta-learning system is penalized every time when the subordinate system does not perform well, and in this way it is forced to improve the subordinate system, which has to learn the current function better (faster and more precisely).

The results have shown that recurrent neural networks are able to learn novel algorithms from a teacher extremely fast, i.e., they can quickly learn quadratic functions seen never before with using only few examples [29].

Learning enhancement can be achieved through exploiting previously learned or through learning in parallel related tasks, because, very likely, a new task shares, to some extent, inductive biases or prior knowledge with other tasks [4].

Humans are able to learn many things from single or few examples ("one-shot" learning) and crucial for that is the ability to extract and use abstract knowledge [4].

Our approach to achieve meta-learning consists of the following:

- instead of a single learning task, we define a family $\mathcal{F}$ (in general, infinitely large) of learning tasks $\mathcal{C}$;
- learning is carried out through two loops - an inner and an outer loop. An inner loop represents a neural network $\mathcal{N}$ (in our case, an LSNN) with a learning algorithm and a set of parameters. An outer loop tries to optimize the parameters of the network $\mathcal{N}$ for a randomly drawn task $\mathcal{C}$ from $\mathcal{F}$;
- outer loop uses some optimization algorithm, for example, BPTT (BackPropagation Through Time), ES (Evolution Strategies), SA (Simulated Annealing), GA (Genetic Algorithm). The goal is to integrate performance evaluations from different tasks $\mathcal{C}$ of the family $\mathcal{F}$, since it enhances learning, as already stated.

3 On Symbolic Computation

Higher-level mental processes, binding together thoughts, experiences and senses, make use of, among other concepts, working memory [30]. Of course, different models of working memory module are proposed and all of them are trying to make an efficient use of it, to integrate it into existing models of networks and use it in many different aspects, for example, LTL setup. With spiking neural networks, it is even more challenging.

On the other side, there are also attempts to make cognitive architectures. Gary Marcus discusses it and sets some concepts which are important for symbolic computation in general.

Through this chapter we will give a short overview of these concepts - working memory from a perspective of neuroscience, concepts proposed by Marcus in his book *"The Algebraic Mind - Integrating connectionism and cognitive science"* (see [7]) and some of the arcitectures proposed to make use of these ideas. We will also mention Neural Turing Machines, also referred to as Memory-Augmented Networks, and how they can be used for "one-shot" learning, as they are similar to our work and since through our experiments we use an LSNN (spiking neural network with LSTM-like units, i.e., memory units) in LTL setup.

3.1 Working Memory

Working memory can be considered to be the most significant achievement of human mental evolution. It refers to the temporary maintenance and manipulation of information for short periods of time (on time scale second to minutes). Tasks such as looking for a lost item (for example, car keys), conversation or driving to work show us that working memory interacts closely with other functions/systems - motor and premotor systems (immediate goals and possible actions to achieve them), cognitive functions (perception, language comprehension and production, thought and intentionality), attention, reasoning, goal-directed behavior and so on [31, 32]. Humans are able to store, retrieve and manipulate symbolic information. Performing mental arithmetic is an example of using symbolic information. It requires storing a string of numbers representing operands and a result of performed subcomputation (for example, the sum of one addition), while calculating the next one [32].

3.2 Related Work

Neural Turing Machines (NTM) are neural networks extended by an external memory, where the interaction between the modules is achieved via attentional processes. With an analogues design as in Turing machines or in systems with Von Neumann architecture and trained with gradient descent,

these architectures are able to learn simple algorithms such as associative recall, copy and sort operation. This architecture reminds a lot to a working memory system, since it makes use of rule-based manipulation (here, simple programs) of stored information (the arguments of these programs). Another similarity to biological working memory system is the use of attention mechanism. The NTM arcitecture has a so-called attentional controller, which makes it possible to implement reading from and writing to memory operations in a selective manner. The advantage of this model is its ability to use working memory content to learn necessary tasks instead of having a fixed set of procedures [33].

One-shot learning using Memory-Augmented Neural Networks A traditional neural network trained using gradient-based algorithms requires an extensive iterative training. Difficulties arrise from the fact that encountering new data forces these models to inefficiently relearn their parameters, so the new information can be adequately incorporated in the existing model, togehter with the previous knowledge. In contrast, humans can learn quickly from just a few examples. Architecture used in this paper combines memory-augmented neural networks, more precisely, previously described NTMs, and a meta-learning setup by Hochreiter, Younger and Conwel, as explained in Section 2.5. The tasks they demonstrate are a regression task with Gaussian process with fixed hyperparameters and a classification task on Omniglot. Meta-learning setup with a delay in presenting labels of data samples forces this network to use memory module (for store and retrieval of data presented in the previous steps and binding sample-class information so it can be reused) [34].

3.3 On Cognitive Architectures

According to Gary Marcus, symbols can represent *categories* (e.g., CAT), *variables* (e.g., x), *computational operations* (e.g., '+', '-', concat, compare) and *individuals* (e.g., Felix). He sees context-independent representations of categories, which multilayer perceptrons have, as symbols and argues that the mind is a system that represent variables and operations over variables, structured representations and a distinction between kinds and individuals. One possible way to implement this system physically is to use a set of buckets, where the bucket's contents indicates the instantiation of a given bucket. Computers already use binary registers (bits) that correspond to buckets being either full or empty and operations which they perform are defined in parallel over these sets of bits. The next question is how a given input variable is represented - using one node or a set of nodes, namely, the number of input units allocated for representing each input variable.

Nevertheless, each variable should correspond to an anatomically defined region or register. For example, the symbol "black" in concepts "black cat" or "black sedan" should apply equally, even though categories animals and

cars in this particular case may be represented in widely separated regions of the cortex. In Dynamically Partionable Autoassociative Neural Networks (DPANN) model [35], a given symbol corresponds to a global attractor state of the dynamics of a large-scale network that links many regions of the brain. Information flow between particular sets of registers is achieved through turning on and off subsets of synapses, i.e. gating [7, 8].

Biologically Inspired Cognitive Architectures (BICA) challenge is an attempt to integrate "neural-level" and "cognitive-level" approach. Since, in general, symbolic representations are superior concerning their interpretability, direct control, coding and extraction of knowledge are some of the difficulties to tackle [36].

ACT-R ("Adaptive Control of Thought—Rational") is an cognitive architecture which models higher-level cognitive processes and possibly could make use of DPANN binding system for production rule matching operations, as suggested in [8].

Vector-symbolic architectures (VSAs) [10]. These architectures use high-dimensional vectors to represent the symbolic concepts encoded by activity patterns of a group of neurons and a defined set of mathematical operations. Both the variables and potential role filters (for example, subject, object or verb) are represented as vectors. Compositional process typical for symbol manipulation is then implemented through some mathematical transformation of these vectors, for example, multiplication of two vectors, for example, *subject* and *cat*, will provide the activity pattern for this binding, assigning cat to the role of subject [8].

4 Implementation

In this chapter we will present relevant implementation details. We will describe neuron and network model and training procedure used in our experiments. Variations and specifics for the experiments will be given in Chapter 5.

4.1 Neuron Model

In all experiments presented through this work, we used spiking neural network with recurrent connections. This network consisted of leaky integrate-and-fire (LIF) neurons, which are modelled as adaptive leaky integrate-and-fire (ALIF) neurons. Such a network is referred to as an LSNN network. As explained in Section 2.3, a neuron j fires when its membrane potential crosses a threshold voltage $b_j(t)$. Adaptive LIF neurons change their dynamics through learning. Their threshold values change at each spike, i.e., threshold of each neuron j is increased by a constant β when neuron j fires and then decays back to a baseline value b_j^0.

Dynamics of these neurons can be described as follows:

$$\tau_m \frac{du_j}{dt} = -u_j(t) + R_m I_j(t), \tag{11}$$

$$\tau_{a,j} \frac{db_j}{dt} = b_j^0 - b_j(t), \tag{12}$$

where τ_m represents the membrane time constant, $u_j(t)$ is the membrane potential, R_m is the membrane resistance, $\tau_{a,j}$ is the adaptation time constant, $b_j(t)$ the threshold and b_j^0 the baseline threshold value.

Neurons are stimulated by synaptic current $I_j(t)$, which is defined as the weighted sum of spikes from all connected neurons (external inputs and other neurons of the network):

$$I_j(t) = \sum_i W_{ji}^{in} x_i(t-d) + \sum_i W_{ji}^{rec} z_i(t-d), \tag{13}$$

where terms W_{ji}^{in} and W_{ji}^{rec} represent synaptic weights for the input and recurrent neurons, respectively, x_i and z_i represent spike trains of neuron j received from the input and recurrent populations, respectively, and term d denotes a synaptic delay.

Our experiments were performed in discrete time, with discretized steps $dt = 1ms$ and the equations then become:

$$u_j(t+dt) = \alpha u_j(t) + (1-\alpha) R_m I_j(t) - b_j(t) z_j(t), \tag{14}$$

$$b_j(t+dt) = \rho_j b_j(t) + (1-\rho_j) z_j(t), \tag{15}$$

where $\alpha = \exp(-\frac{dt}{\tau_m})$, the current $I_j(t)$ represents a weighted sum of all incoming spikes, $\rho = \exp(-\frac{dt}{\tau_{a,j}})$ and $z_j(t)$ represents incoming spike trains. The reset of membrane voltage is implemented through the term $b_j(t)z_j(t)$.

In all experiments, it was sufficient to use the same parameters for LIF/ALIF neurons and they are listed in Tables 1 and 2.

Parameter	**Value**
maximum firing rate	200 Hz
threshold b_j	0.03 V
dt	1 timestep
refractory steps	5 timesteps
delay steps d	5 timesteps
τ_m	20 ms
dampening factor	0.4 or 0.3

Table 1: **Parameters for all spiking neurons.**

Parameter	**Value**
β	1.6
$\tau_{a,j}$	1-1000ms or 1-8000ms

Table 2: **Additional parameters for adaptive spiking neurons (ALIF).**

4.2 Network Model

Proportion of 40% of all neurons in this recurrently connected network are adaptive, others are referred to as regular spiking neurons, i.e., their thresholds are not adaptive, but rather constant. Adaptive neurons are denoted as a population A, whereas regular neurons are denoted as a population R, as depicted in Figure 5.

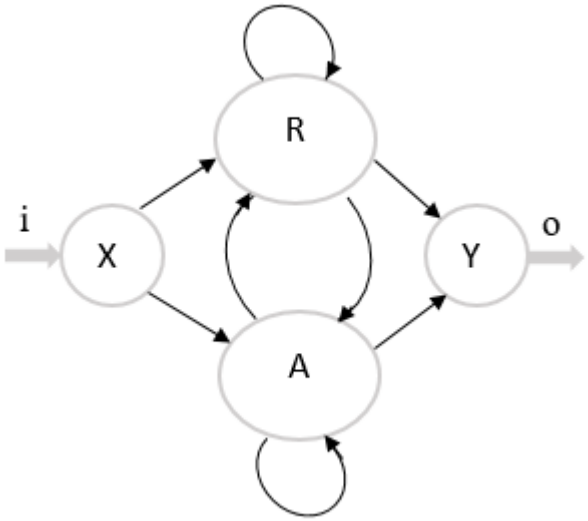

Figure 5: **An LSNN network.** X, Y, R, A denote populations of neurons - input, output, regular and adaptive populations, respectively. Connections from one population to another population or to itself are indicated by arrows. All inputs that have to be encoded by the population X to spike trains are denoted as i, whereas o denotes the output which has to be read out from the population Y (or in the simplest case, a single neuron).

A population of input neurons X represents external inputs to both populations R and A in form of spike trains. For example, through all the experiments we have two real-valued variables x and y, both $x, y \in [-1, 1]$ whose values are encoded by a population consisting of 200 neurons (100 neurons for one real-valued variable), and whose spike trains are fed in as an input to this network. Depending on experiment performed, input population X can be extended in order to encode some other relevant information. Output Y is always a result of computation performed by this network. It is a linear readout neuron, whose inputs are mean firing rate per step (time windows of 20 or $40ms$) of all neurons from populations R and A. For the first experiment, duration of step was $20ms$ and mean firing rates were calculated for these time windows of $20ms$. In our second and third experiment, duration of step was $40ms$, and mean firing rates rates were calculated for the time windows of last $20ms$ of every step.

4.3 Input Encoding

To perform our experiments, it was necessary to encode analog values into spike trains. Analog values were real numbers from the input range [-1, 1] and encoded by an input population X of spiking neurons in the following way: each analog value was encoded by 100 input neurons, $m_1, m_2, ..., m_{100}$.

Values from the input range were equally distributed and assigned to each of these 100 neurons, which had a Gaussian response with particular mean on that analog value and a constant standard deviation of $\sigma = 0.002$. Firing rate of a neuron i is given by:

$$r_i = r_{max}\, exp(-\frac{(m_i - z_i)^2}{2\sigma^2}), \tag{16}$$

where $r_{max} = 200Hz$, m_i is the value for which neuron i is responsible and z_i is some value from the input range.

For some experiments, it was necesarry to give additional information, for example, an identifier of a function or a rule for symbolic expression. To encode these additional input, we used one-hot encoding, with as many neurons as indicated by the length of one-hot vectors. All of these neurons always fire with a firing rate of $2Hz$, except for the one (exactly there where 1 is indicated in one-hot vector), which fires with a firing rate of $200Hz$.

4.4 Generation of Nonlinear Functions for LSNN to Learn

A simple two-layer feed-forward neural network with two input neurons and one hidden, fully connected layer and an output sigmoid neuron was used to generate nonlinear functions, which the LSNN had to learn. This architecture is depicted in Figure 6 and we call it a target network (TN). The set of parameters describing this TN is given in Table 3.

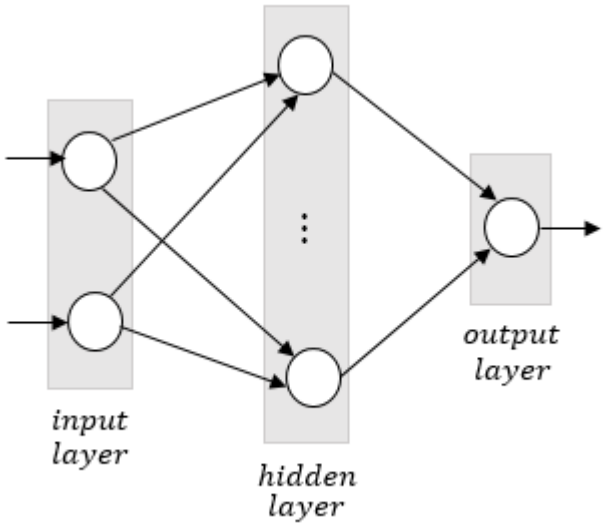

Figure 6: **A target network, TN.** It consists of sigmoidal neurons. Layers are fully connected.

Parameter	Value
number of input neurons	2
number of neurons in the hidden layer	10
number of output neurons	1
use bias	True
input range	[-1, 1]
output range	[0, 1]

Table 3: **Parameters of Target Network.**

All neurons (except for input neurons) have weights and biases as parameters, which are uniformly drawn from range [-1, 1]. For example, the total number of parameters of one TN, which defines one nonlinear function which LSNN has to learn is:

$$n_{coefs} = (n_{inputs} * n_{hidden} + n_{hidden}) + (n_{hidden} * n_{output} + n_{output}). \quad (17)$$

Outputs are resized to values in range $[0, 1]$.

4.5 Training Procedure

In a recurrent neural network, gradients are propagated through time. An unfolded recurrent network is trained using Backpropagation Through Time (BPTT) algorithm. Term unfolding of a recurrent neural network indicates that its computational graph represents spatially unfolded (unrolled) time steps. Since gradients are propagated over many steps, they tend to either vanish or explode. To overcome this problem, we can use LSTM (Long-Short Term Memory) units. The idea behind is to create paths through time that neither vanish nor explode [5]. For spiking neural networks, computing gradients differs from the previously described case. The reasons for this are:

- outputs of spiking neurons are non-differentiable,
- gradients need to be propagated through continuous time or
- in case of time being discretized, gradients need to be propagated through many steps

and hence BPTT algorithm cannot be used. Instead, one can use a pseudo-derivative, as proposed in [37, 21], for binary neurons, or as proposed in [16] a dampened pseudo-derivative:

$$\frac{dz_j(t)}{dv_j(t)} := \gamma \, max\{0, 1 - |v_j(t)|\}, \quad (18)$$

$$v_j(t) = \frac{u_j(t) - b_j(t)}{b_j(t)}, \tag{19}$$

where $v_j(t)$ represents the normalized membrane potential and γ is a dampening factor, typically set to 0.3 or 0.4.

Training was implemented through iterations depicted in Figure 7. Update of weights was performed after every iteration, which consisted of a batch of 10 episodes. In each episode there were n steps and every step was 20 or $40ms$ long.

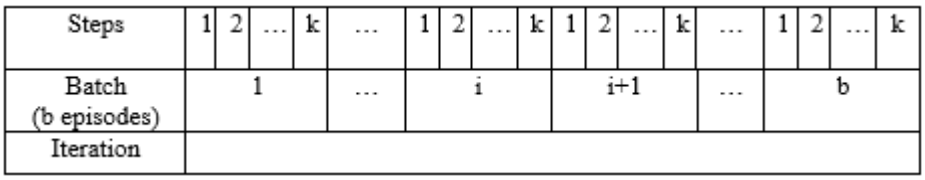

Figure 7: **Illustration of training and test realization.**

In particular, each episode represents one function or symbolic expression that our network had to learn, and each step was one concretization of that function or symbolic expression.

Weights are initialized randomly according to normal, Gaussian distribution, with zero-mean and unit-variance, $\mathcal{N}(0, 1)$ and rescaled with a factor $1/\sqrt{num_{rec}}$, where num_{rec} is the number of recurrently connected neurons (all neurons of the LSNN network). All biases are initialized with the same constant value, zero.

As an optimization algorithm, Adam optimizer (Adaptive Moment Estimation) was used, with learning rate $lr = 0.001$. Training was performed for 5000 iterations, testing for 100 iterations. The loss function was the Mean Squared Error (MSE) of the network's predictions for one iteration. The linear readout neuron was used to read network's predictions for each step, calculated as an average over spikes over $20ms$ time window. The final loss was calculated as $L = MSE + 30R$, where the term R represents rate regularization term, introduced to maintain a neuron firing rate of $20Hz$, and it was defined as MSE(average firing rate of the LSNN neurons $-$ target rate of $20Hz$).

5 Experiments

In this chapter we will describe performed experiments. As already discussed earlier, in the setup from Hochreiter (see [29]), meta-learning is used and we use it in a similar way. We start with learning nonlinear functions using LSNN. Once this task is successful, we move to our next, main experiment - symbolic computation. Finally, our third experiment aims to test if network is still able to perform computation even if we stop showing the symbolic expression at some step.

5.1 Learning Nonlinear Functions

In these experiments, a two-layer feed-forward neural network (target network, TN), consisting of sigmoid neurons, is used to generate nonlinear functions which the LSNN had to learn. The architecture of this target network is illustrated in Figure 6 and its parameters are given in Table 3.

Given two inputs, $x_1, x_2 \in [-1, 1]$, outputs are obtained as $y = f_i(x_1, x_2)$, $i = 1, \ldots, 100$. Each nonlinear function f_i has fixed parameters, which represent weights and biases in the target network (Expression 17), hence denoted as a finite family of functions. We use 100 different nonlinear functions in our experiments.

Training was performed through 5000 iterations, consisting of batches of 10 episodes. Each episode represented one nonlinear function and training examples of one episode, for a concrete function f_i, were presented in 500 steps. Each step was one application of the function f_i. Dataset for this experiment was generated using TNs, whose architecture was described in Section 4.4 and had two inputs and one output. So, concrete values x_1, x_2, which are randomly chosen for each step, were inputs for the TN which corresponded to that particular f_i, and the output of TN, $f_i(x_1, x_2)$, we denote as $C(x_1, x_2)$.

5.1.1 Learning Nonlinear Functions Using Previous Targets

First experimental setup was to give the network inputs x_1 and x_2, to which some f_i was applied and for which we know $C(x_1, x_2)$. We have third input for enabling supervised learning, but we do not present targets immediately, but only after the network makes a guess, i.e., in the next step this third input, the result of previous application of f_i to arguments (x_1, x_2) will be given and we denote it as $C'(x_1, x_2)$. Since network should learn nonlinear mappings f_i, in the very first step (step 1 of all steps), we let network guess the output and we set third input to zero. More precisely, inputs in every step are triples $(x_1, x_2, C'(x_1, x_2))$. This experimental setup was proposed by Hochreiter in [29], using RNNs and LSTMs.

5.1.2 Learning Nonlinear Functions Using Symbolic Identifiers

Each nonlinear function represents a "computational program" which LSNN has to learn. In addition, we also provide a symbolic identifier for each of these computational programs. The LSNN should learn to apply each of the computational programs through the symbolic identifier. That would imply that each learned computational program is encoded and stored in the "working memory" of the LSNN and that it is possible to reproduce the working memory content from the identifier.

We use finite family of nonlinear functions, generated by the TNs, hence a symbolic identifier is an identifier of a function from this finite family, where each function has fixed coefficients.

5.1.3 Comparison and Results

The LSNN network for both experiments consisted of 250 recurrently connected neurons. For the needs of the first experiment (using previous targets), input population had 300, since we need to encode 3 real-valued variables to spike trains. For our second experiment, we need to encode x, y and function identifiers, and hence, for the number of functions in finite family of functions, we chose number 100. The input population is then of the same size for both experiments and hence, comparable.

We also wanted to compare it with a linear baseline. It was generated using linear regression with L2-norm regularizer of factor 100, trained using not analog values, but average spiking traces of all neurons (because spike trains are inputs to LSNN). First, an exponential kernel ($20ms$ width and time constant $20ms$) was applied to obtain spike traces from spiking activity of input neurons (representing variables x_1 and x_2) and then we calculate mean values for every step. We train the linear regressor using 90% of randomly chosen points (in one iteration, when calculating MSE over iterations, or in all points representing particular step, when calculating MSE over steps) and test it on the remaining 10% of all points; we cross-validate it 5 times.

Both experiments perform better than the linear baseline. Figures 8 and 9 show comparison of MSE over iterations and steps for both experiments.

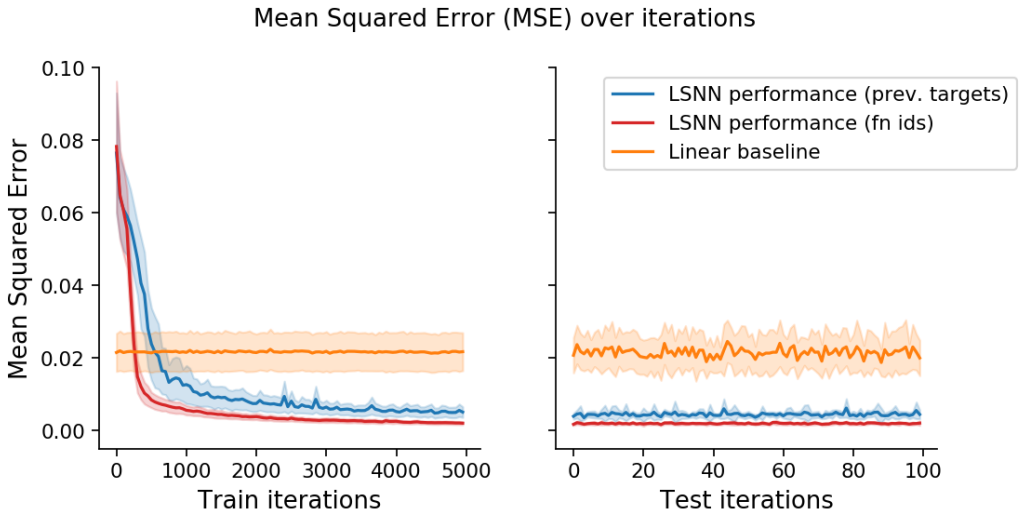

Figure 8: **LSNN performance over training and testing iterations, compared with a linear baseline.** Learning new nonlinear functions was implemented through LTL setup. MSE for LSNN using function identifiers during testing was 0.002, for LSNN using previous targets 0.0045 and for linear baseline 0.0215.

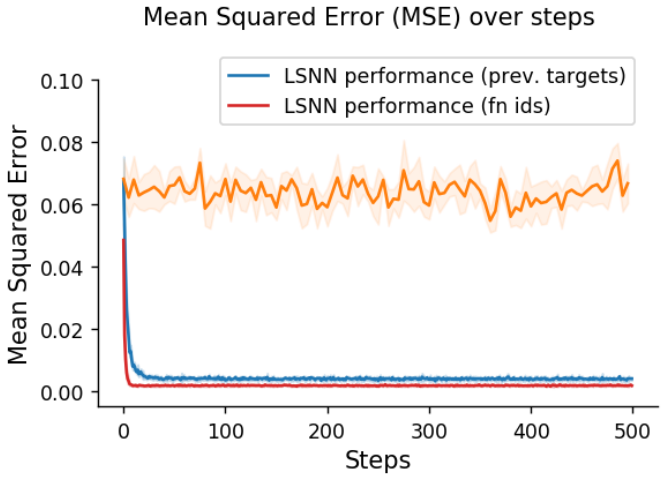

Figure 9: **LSNN performance over steps, compared with a linear baseline.** Performance given here shows the average behavior of the network for nonlinear functions, used in 100 test iterations, each with 10 episodes. MSE for linear baseline here was 0.0639.

We will focus now on experiment with learning using symbolic (function) identifiers, because learning using previous targets is already presented and discussed in [16]. The worst and best performing episodes are shown in Figure 10, with MSEs 0.0034 and 0.0011, respectively.

Since our network learns very quickly, in only few steps, we were interested to see what was the internal model of the network and its progress during learning a new nonlinear function restored by using an identifier, as depicted in Figure 11.

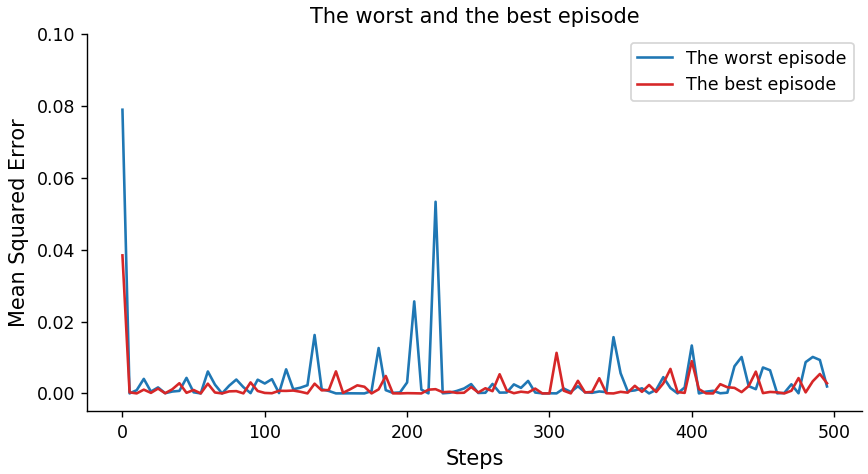

Figure 10: **The worst and best performing episode, learned through identifiers.** Each episode represents a nonlinear function, which LSNN has learned and is able to use later when the identifier is given.

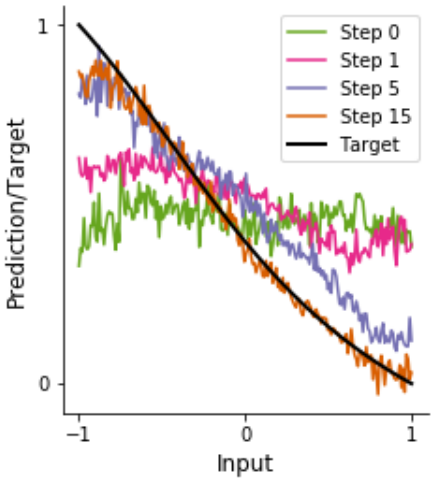

Figure 11: **Progress during learning a new nonlinear function, given through an identifier.** Change of internal model over few steps is illustrated and shows a rapid progress.

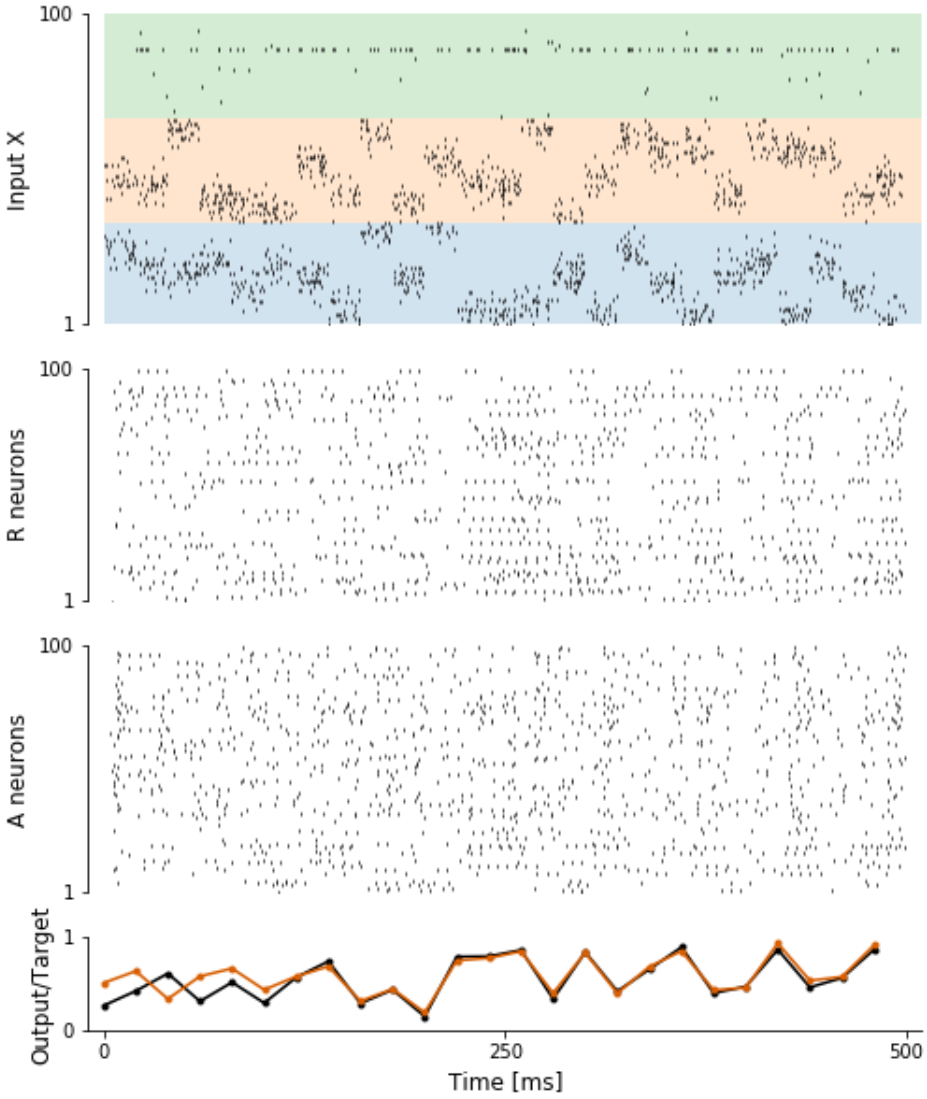

Figure 12: **Spike raster.** Shown is the firing activity of inputs (first subplot), recurrently connected neurons, regular and adaptive (R and A populations, respectively) (middle 2 subplots) and output-target correlation. Population of input neurons X encodes an identifer of a function (first panel of the first subplot) and concrete real values, x_1 and x_2 (middle and bottom panels of the first subplot). Spike activity presented here is for time window $0 - 500ms$, i.e., first 25 steps of an episode.

5.2 Learning Symbolic Expressions

We want to move in direction of calculating symbolic expressions, i.e., to be able to calculate expressions of the form

$$Var_1 \quad Sign \quad Var_2 = Solution, \tag{20}$$

where $Var_1 \in \{x, y\}$, $Sign \in \{+, -, *\}$, $Var_2 \in \{x, y\}$.

Two concrete values for variables x and y are always given, but depending on the concrete expression, it might happen that only one value is needed and used. That happens in cases where $Var1 = Var2$, for example, in expressions $x + x$, $y * y \ldots$ These concrete values are encoded by spike trains of 200 neurons from input population X (100 neurons for each concrete value), as in experiments before. For symbolic expressions, we use one-hot encoding, so we need 7 more neurons to encode it:

$$x|y| + | - | * |x|y$$

Network consisted of 300 recurrently connected spiking neurons, 40% of them were adaptive (population A), the rest were regular spiking neurons (population R). Output was a linear readout neuron, calculating the result of the computation based on mean firing rate for steps of duration $40ms$, but taken only for the second half of this time window. In this way, symbolic expressions were presented to the network longer, leading to slightly more accurate computation, compared to the steps of duration $20ms$ and mean firing rates for the whole time window.

Training was performed for 5000 iterations, whereas testing was perfomed for 100 iterations. In both cases, batches consisted of 10 different episodes and each episode lasted 100 steps of $40ms$. Each episode represented one symbolic expression and in every step concrete values (for x and y) were substituted in symbolic expression. Figure 13 shows the performance of LSNN network during training and testing iterations.

We also observed what happens in each episode. After only few steps, predictions become very good. In fact, network needs few steps to adapt to given expression, to install it and start using. Figure 14 shows performance of LSNN for 100 iterations of testing with batches of size 10. Note that these $100 * 10 = 1000$ episodes, with their 100 steps are overlapped to illustrate the performance over steps.

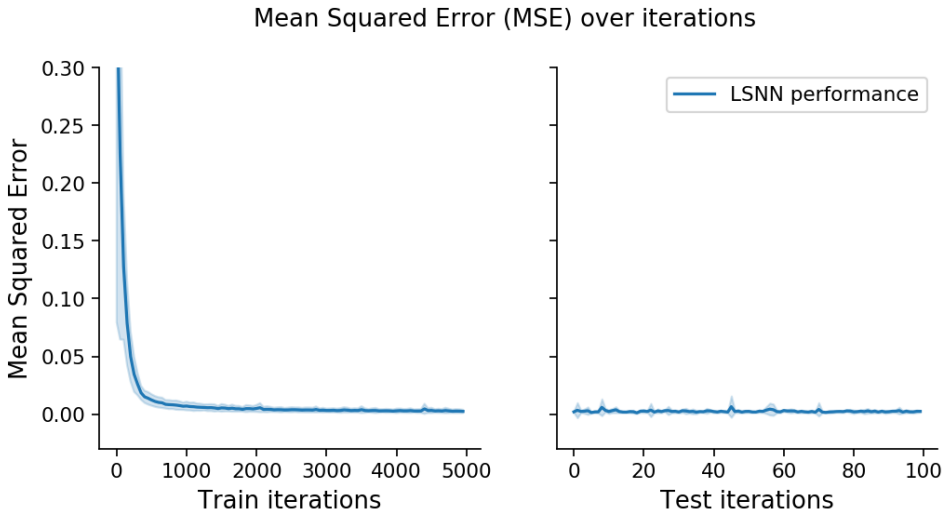

Figure 13: **LSNN performance over training and testing iterations.** Learning new symbolic expressions was implemented through LTL setup. Through a single iteration, consisting of 10 episodes, network is forced to transfer knowledge (accumulate and apply preceding knowledge). MSE for testing iterations was 0.0024.

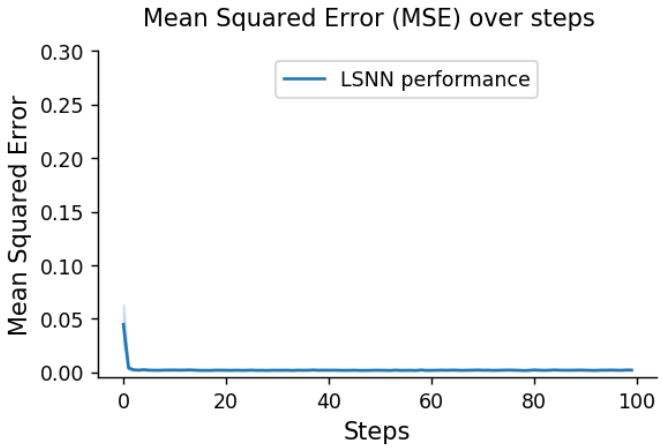

Figure 14: **LSNN performance over steps.** Performance given here shows the average behavior of the network for symbolic expressions, presented in 100 test iterations, each with 10 episodes. After few steps, the network is able to "reuse" the rule for symbolic expression and achieves very good performance for all remaining steps in episode. MSE = 0.0024.

Again, we focus on testing phase and want to inspect two single episodes, the worst and best performing one, where criteria was simply to find maximum and minimum mean value of the episode, skipping first 5 steps. These

two single episodes are shown in Figure 15. Tables 4 and 5 demonstrate few concrete calculation steps for these two episodes. MSE for the worst performing episode (without taking first 5 steps in consideration) was 0.005, whereas for the best performing one was 0.0003, which shows that even the worst performing episode is performing very well.

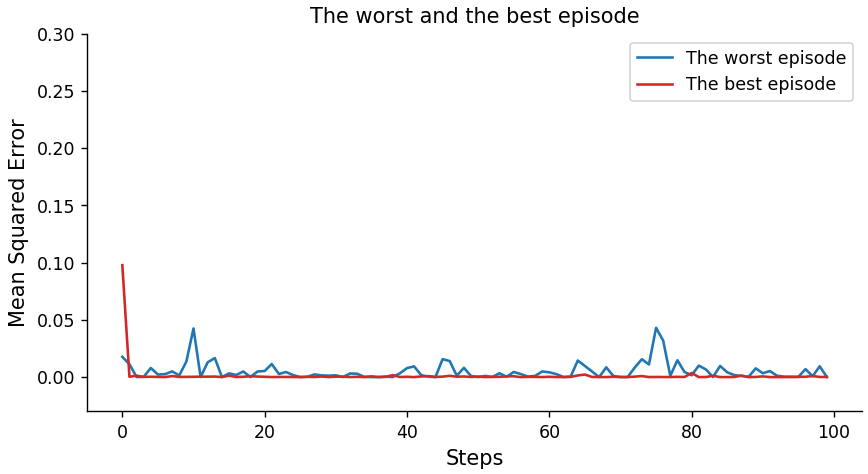

Figure 15: **The worst and best performing episode.** These episodes are chosen from test iterations, based on mean value of all but first 5 steps of MSEs in episode. MSEs for the first 5 steps in each episode are excluded from this analysis, because the network needs few steps to adapt to new symbolic rule and usually makes higher errors there.

Step no.	**Symbolic expression: x + x**	**LSNN output**	**MSE**
1.	-0.6279 + (-0.6279) = -1.2558;	-1.1228;	0.0177;
2.	-0.9994 + (-0.9994) = -1.9988;	-1.8927;	0.0113;
3.	0.5662 + 0.5662 = 1.1324;	1.1249;	0.0001;
4.	-0.0162 + -0.0162 = -0.0324;	-0.0445;	0.0001;
5.	0.5622 + 0.5622 = 1.1243;	1.0350;	0.0080;
50.	0.5866 + 0.5866 = 1.1733;	1.1431;	0.0009;
51.	0.3724 + 0.3724 = 0.7447;	0.7388;	0.0000;
96.	-0.7887 + (-0.7887) = -1.5773;	-1.5915;	0.0002;
97.	0.1002 + 0.1002 = 0.2003;	0.1167;	0.0070;
98.	-0.4471 + (-0.4471) = -0.8942;	-0.9213;	0.0007;
99.	0.8005 + 0.8005 = 1.6010;	1.6983;	0.0095;
100.	-0.7640 + (-0.7640) = -1.5280;	-1.5261;	0.0000;

Table 4: **The worst performing episode, with concrete values for steps** $1-5$**,** $50-51$**,** $96-100$**.** Symbolic expression used in this episode was $z = x + x = 2x$.

Step no.	**Symbolic expression: x - x**	**LSNN output**	**MSE**
1.	-0.8971 - (-0.8971) = 0;	0.3128;	0.0978;
2.	0.3485 - 0.3485 = 0;	0.0153;	0.0002;
3.	-0.2366 - (-0.2366) = 0;	0.0357;	0.0013;
4.	-0.1055 - (-0.1055) = 0;	0.0134;	0.0002;
5.	-0.5302 - (-0.5302) = 0;	-0.0174;	0.0003;
50.	0.0255 - 0.0255 = 0;	0.0100;	0.0001;
51.	-0.4733 - (-0.4733) = 0;	0.0197;	0.0004;
96.	-0.1535 - (-0.1535) = 0;	-0.0131;	0.0002;
97.	-0.9593 - (-0.9593) = 0;	0.0173;	0.0003;
98.	0.5423 - 0.5423 = 0;	0.0307;	0.0009;
99.	-0.1916 - (-0.1916) = 0;	-0.0099;	0.0001;
100.	-0.9174 - (-0.9174) = 0;	-0.0120;	0.0001;

Table 5: **The best performing episode, with concrete values for steps** $1-5$**,** $50-51$**,** $96-100$**.** Symbolic expression used in this episode was $z = x - x = 0$.

Through this task, the network had to learn different expressions, i.e., functions, which can be grouped in the following way:

- expressions $z = x - x$ and $z = y - y$, where the target function is a horizontal line $z = 0$,
- expressions $z = x + x = 2x$ and $z = y + y = 2y$, where the target function is a linear function (with a slope),
- expressions $z = x * x = x^2$ and $z = y * y = y^2$, where the target function is a quadratic function, i.e., a parabola,
- expression $z = x - y$, $z = y - x$, $z = x + y$, where the target functions are different planes in a 3D space,
- expression $z = x*y$, where the target function is a hyperbolic paraboloid,

and they are presented in Figures 16 - 22.

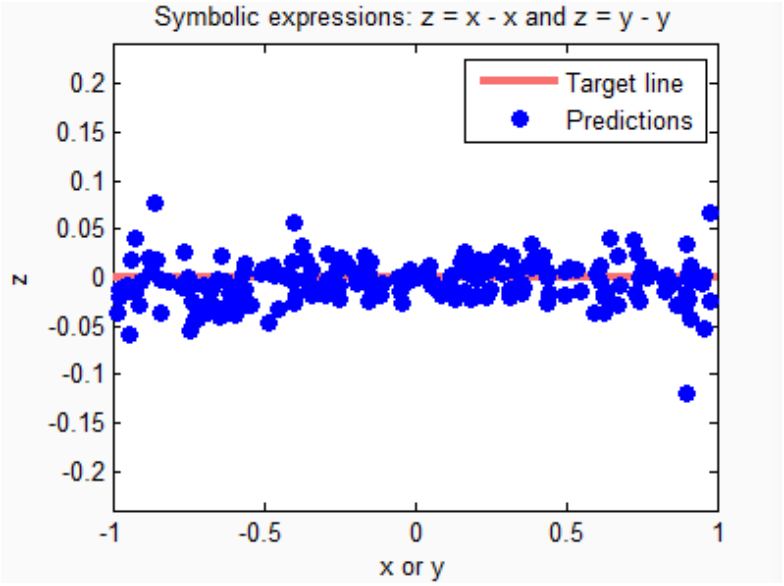

Figure 16: **Symbolic expression subtraction, where both operands are the same variable.** Here, symbolic expressions $z = x - x = 0$ and $z = y - y = 0$ from two episodes are presented together, since for both expressions the target function is a horizontal line, $z = 0$. In this concrete case, most predictions are in the range $[-0.05, 0.05]$.

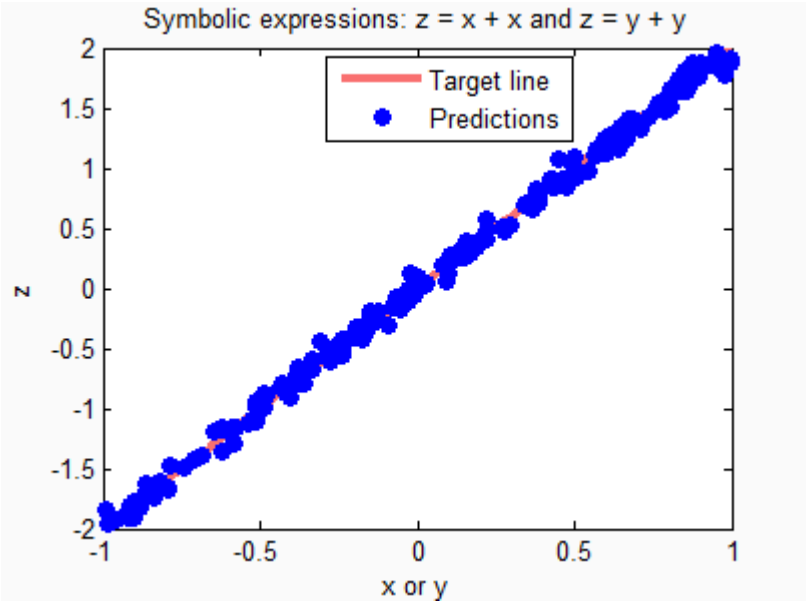

Figure 17: **Symbolic expression addition, where both operands are the same variable.** Here, symbolic expressions $z = x + x = 2x$ and $z = y + y = 2y$ from two episodes are presented together, since for both expressions the target function is a linear function.

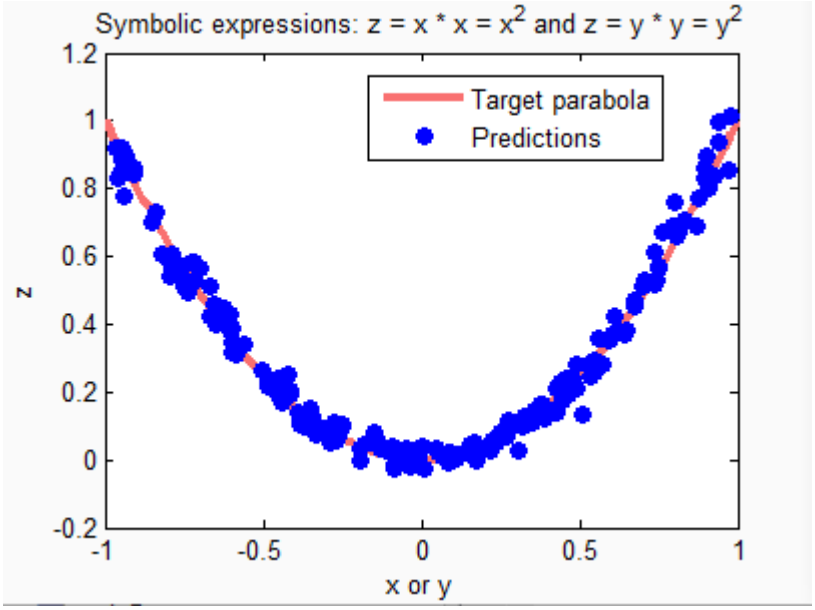

Figure 18: **Symbolic expression multiplication, where both operands are the same variable.** Here, symbolic expressions $z = x * x = x^2$ and $z = y * y = y^2$ from two episodes are presented together, since for both expressions the target function is a parabola.

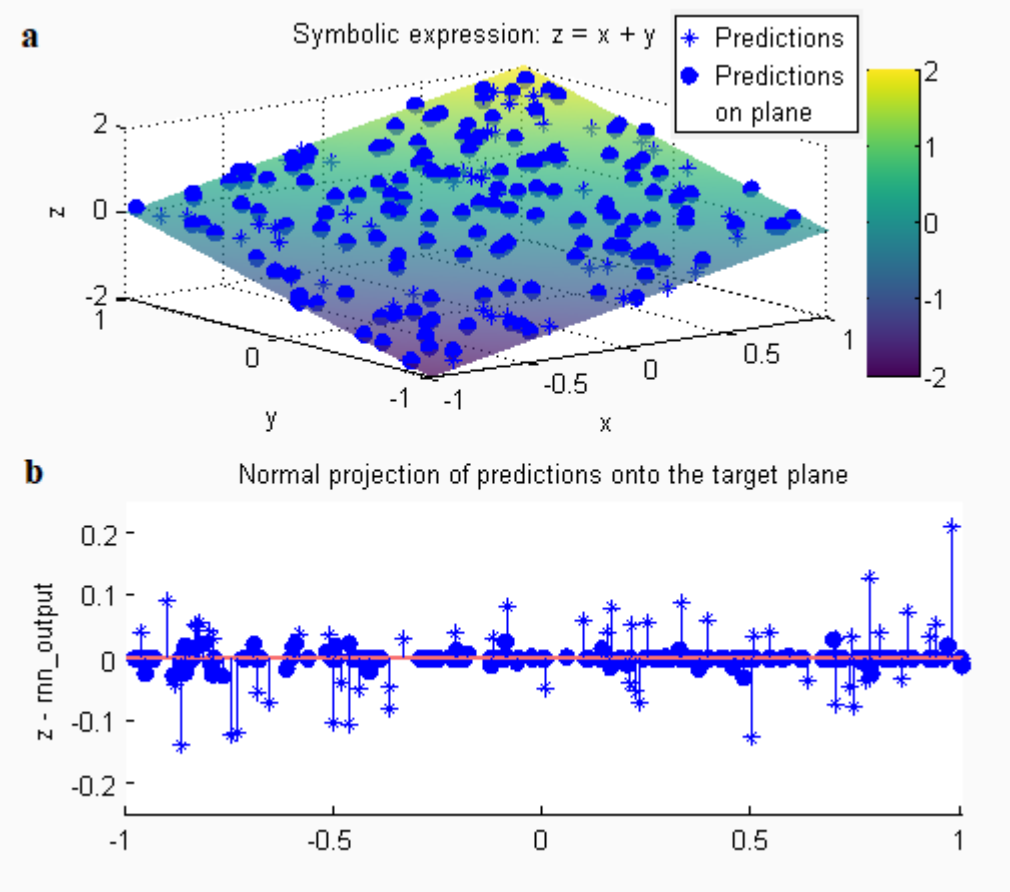

Figure 19: **Symbolic expression** $z = x + y$**.** Predictions from two episodes are presented here together, since addition is a commutative function. **a** Target plane for $x, y \in [-1, 1]$, consequently, $z \in [-2, 2]$. Predictions on the plane, marked with dots, are points which are ϵ-close to the plane, $\epsilon = 0.03$. All other predictions are marked with star-sign. **b** Distances of all points to the target plane. These normal projections, illustrated by vertical lines, correspond to predictions from a. Note that vertical lines without upper bound have distances greater than 0.2.

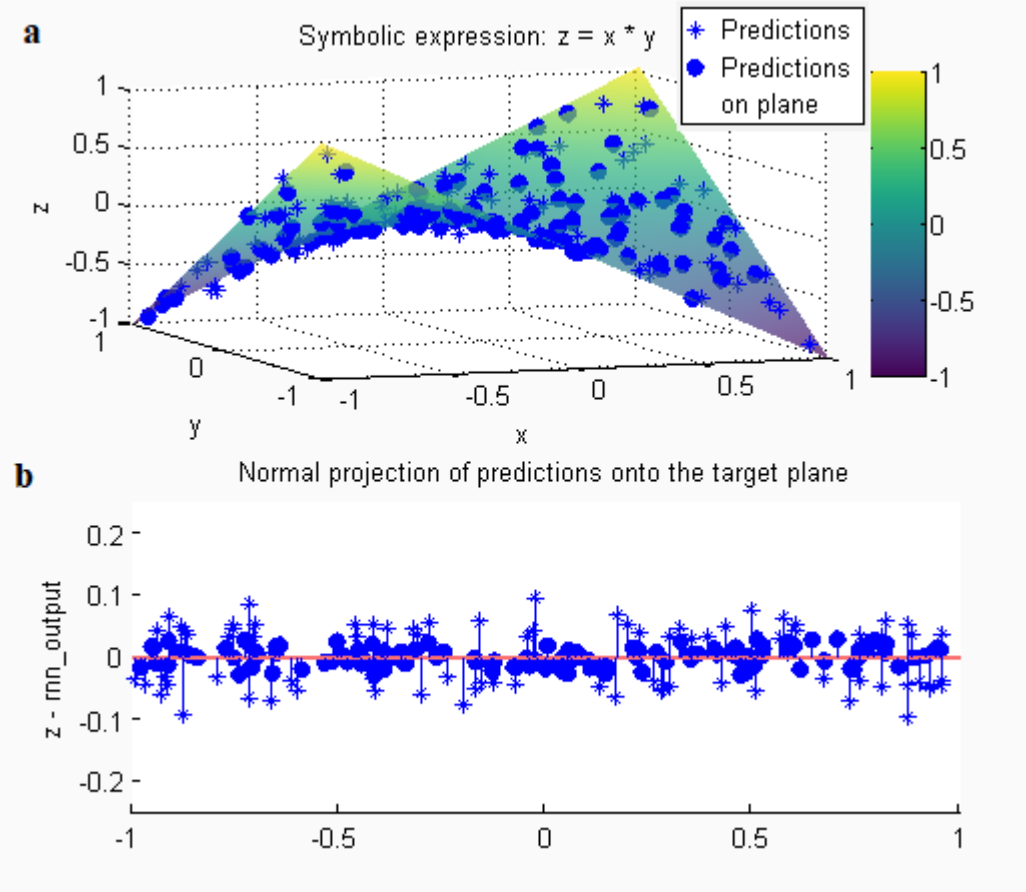

Figure 20: **Symbolic expression** $z = x * y$**.** Predictions from two episodes are presented here together, since multiplication is a commutative function. **a** Target hyperbolic paraboloid for $x, y \in [-1, 1]$, consequently, $z \in [-1, 1]$. Predictions on the plane, marked with dots, are points which are ϵ-close to the hyperbolic paraboloid, $\epsilon = 0.03$. All other predictions are marked with star-sign. **b** Distances of all points to the target hyperbolic paraboloid. These normal projections, illustrated by vertical lines, correspond to predictions from a.

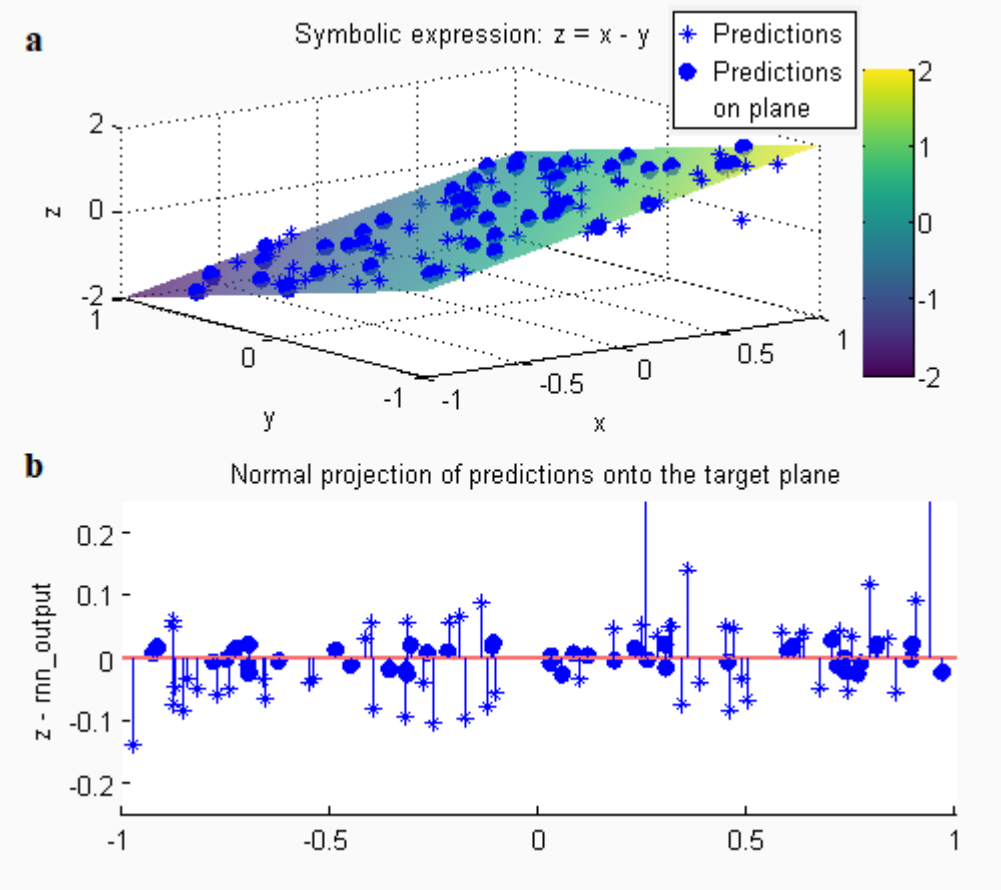

Figure 21: **Symbolic expression** $z = x - y$**.** Subtraction is not a commutative operation. **a** Target plane for $x, y \in [-1, 1]$, consequently, $z \in [-2, 2]$. Predictions on the plane, marked with dots, are points which are ϵ-close to the plane, $\epsilon = 0.03$. All other predictions are marked with star-sign. **b** Distances of all points to the target plane. These normal projections, illustrated by vertical lines, correspond to predictions from a. Note that vertical lines without upper bound have distances greater than 0.2.

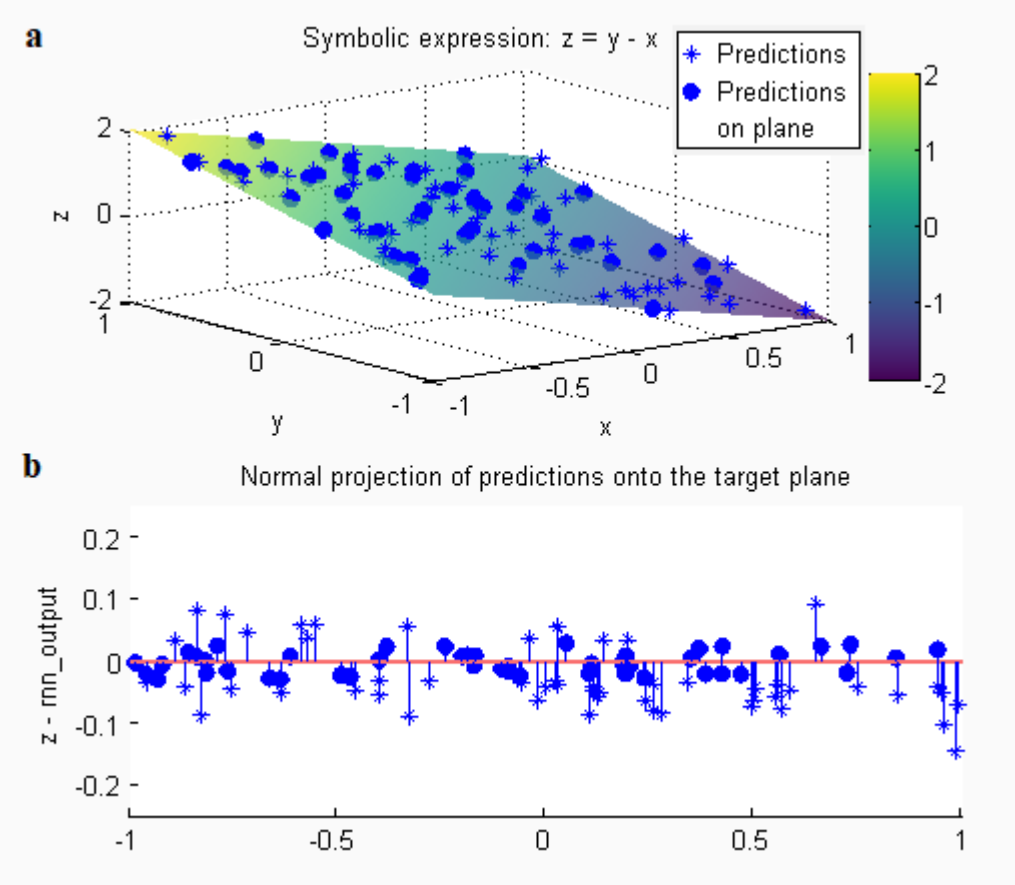

Figure 22: **Symbolic expression** $z = y - x$**.** Subtraction is not a commutative operation. **a** Target plane for $x, y \in [-1, 1]$, consequently, $z \in [-2, 2]$. Predictions on the plane, marked with dots, are points which are ϵ-close to the plane, $\epsilon = 0.03$. All other predictions are marked with star-sign. **b** Distances of all points to the target plane. These normal projections, illustrated by vertical lines, correspond to predictions from a.

Finally, we show spike raster for one chosen episode.

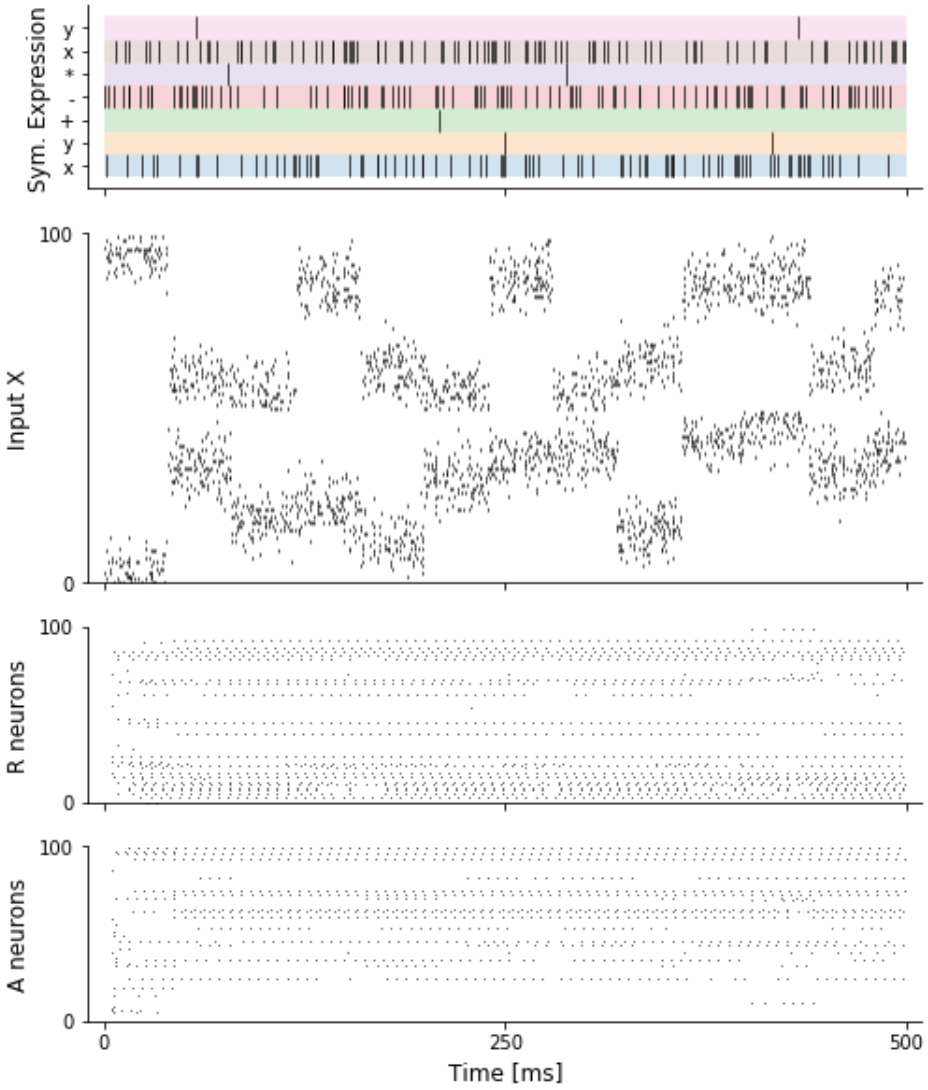

Figure 23: **Spike raster.** Shown is the firing activity of inputs (upper 2 subplots) and recurrently connected neurons, regular and adaptive (R and A populations, respectively) (lower 2 subplots). Population of input neurons X encodes concrete real values, x and y, and symbolic expression. Presented symbolic expression in this episode was $x - x$. Spike activity presented here is for time window $0 - 500ms$.

5.3 Remembering Symbolic Expressions

Through our previous experiments we have seen that in each episode, after few steps, the network is able to restore and reuse the symbolic rule (expression or identifier) and achieve very low errors. Hence we were motivated to test memory capabilities of the LSNN network. How much does the network rely on given symbolic expression? Is it able to correctly perform

computation even when the neurons responsible for symbolic rule do not fire anymore? Here we perform symbolic computation as in previously described experiment, with one difference - we stop showing symbolic expression after 20 steps.

MSE over iterations, illustrated in Figure 24 shows that the learning is less stable, compared with the previous experimental setup, with occasional jumps, but still with a convergence and a low error during testing.

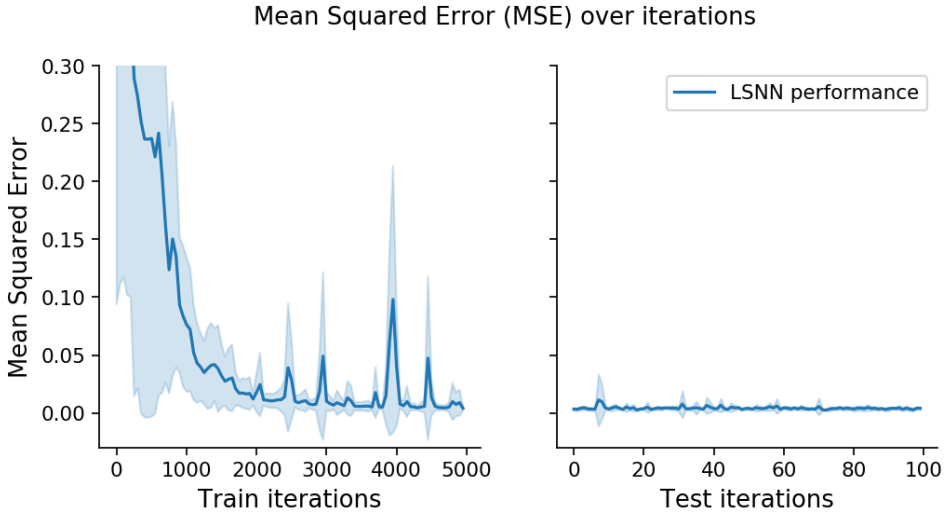

Figure 24: **LSNN performance over training and testing iterations**. The task was to remember and use the symbolic expression given in the beginning of every episode. Learning was implemented throughout LTL setup. Although MSE over training is occasionally unstable, performance during testing shows low errors.

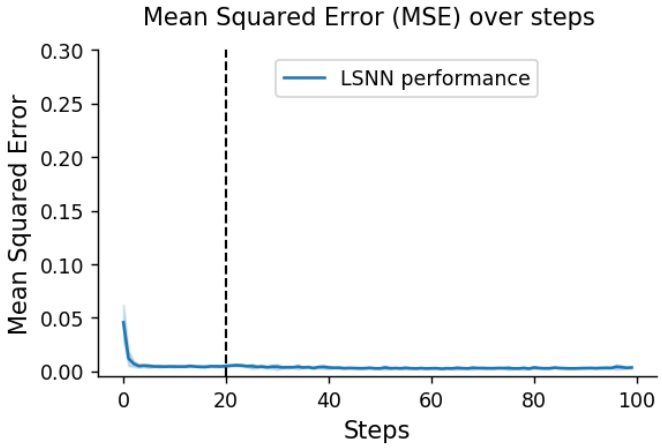

Figure 25: **LSNN performance over steps.** Performance given here shows the average behavior of the network for symbolic expressions, presented in 100 test iterations, each with 10 episodes. Symbolic expressions are shown only during first 20 steps in each episode, illustrated here with a vertical dashed line. After that, the network is still able to perform computation and achieves very good performance for all remaining steps in the episode. MSE = 0.0052.

There are 100 steps in each episode, but the symbolic expression was given only for the first 20 steps. For the remaining 80 steps, network should remember to use the previously given symbolic expression. In this way, adaptive neurons should use their longer term memory. In some cases, it happens that after stopping showing formula, performed computation gets worse. Exactly that happens in the worst performing episode for this experiment, with MSE = 0.078 over steps, excluded first 5 steps, illustrated in Figure 26, together with the best performing episode, for which MSE was 0.001 (excluded first 5 steps). ome steps from these two episodes are shown in Tables 6 and 7.

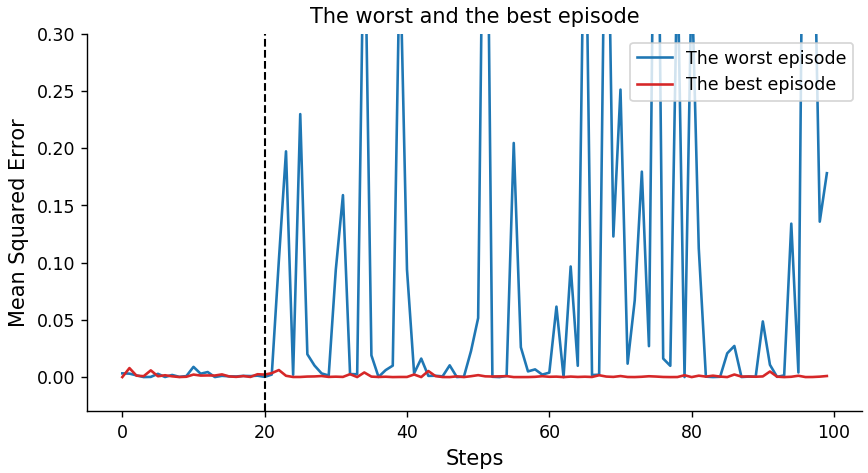

Figure 26: **The worst and best performing episode.** These episodes are chosen from test iterations, based on mean value of all but first 5 steps of MSEs in episode. MSEs for the first 5 steps in each episode are excluded from this analysis. A vertical dashed line denotes the moment when symbolic expressions are stopped being shown to the network.

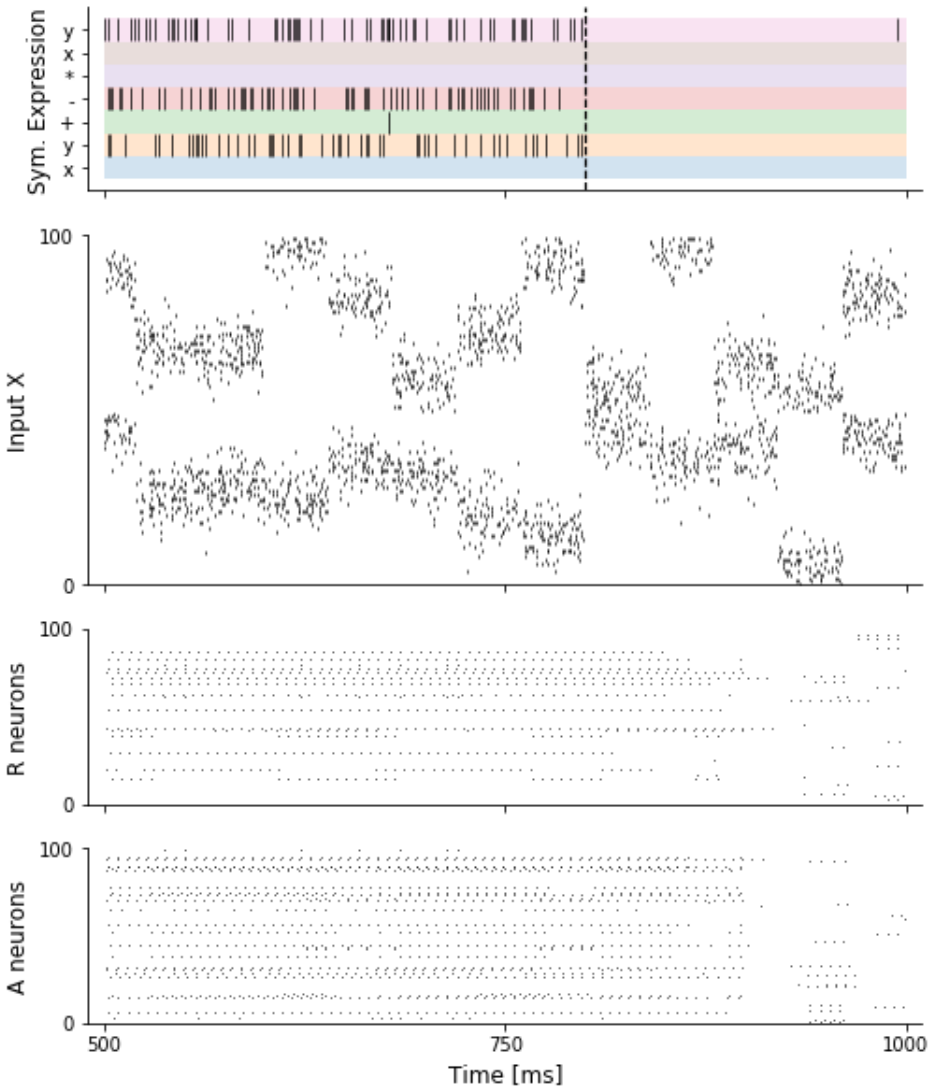

Figure 27: **Spike raster.** Shown is the firing activity of inputs (upper 2 subplots) and recurrently connected neurons, regular and adaptive (R and A populations, respectively) (lower 2 subplots). Population of input neurons X encodes concrete real values, x and y, and symbolic expression. Presented symbolic expression in this episode was $y - y$. Spike activity presented here is for time window $500 - 1000ms$, because one can clearly see the moment of stopping showing the symbolic expression, marked with a dashed line in the first subplot.

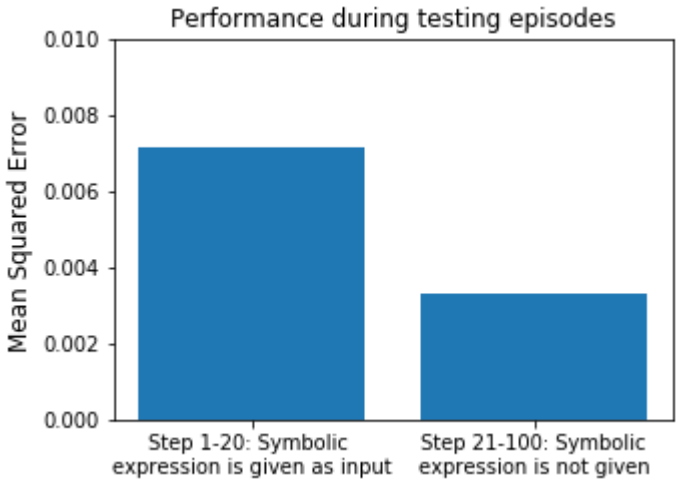

Figure 28: **MSE over "learning" and "testing' 'steps.** "Learning" here means that symbolic expressions are given as input, and "testing" that the network had to rely on its memory and use previously given symbolic expressions.

Step no.	**Symbolic expression: y - y**	**LSNN output**	**MSE**
1.	0.5296 - 0.5296 = 0;	-0.0571;	0.0033;
2.	0.5278 - 0.5278 = 0;	-0.0550;	0.0030;
3.	0.6400 - 0.6400 = 0;	-0.0387;	0.0015;
4.	0.6567 - 0.6567 = 0;	-0.0031;	0.0000;
5.	-0.4489 - (-0.4489) = 0;	-0.0124;	0.0002;
50.	-0.6562 - (-0.6562) = 0;	0.1513;	0.0229;
51.	-0.7321 - (-0.7321) = 0;	0.2272;	0.0516;
96.	-0.2836 - (-0.2836) = 0;	0.0632;	0.0040;
97.	-0.9507 - (-0.9507) = 0;	0.8469;	0.7173;
98.	-0.8598 - (-0.8598) = 0;	0.6858;	0.4703;
99.	0.6516 - 0.6516 = 0;	0.3686;	0.1359;
100.	0.6617 - 0.6617 = 0;	0.4222;	0.1782;

Table 6: **The worst performing episode, with concrete values for steps** $1-5$**,** $50-51$**,** $96-100$**.** Symbolic expression used in this episode was $z = y - y$.

Step no.	Symbolic expression: y - y	LSNN output	MSE
1.	-0.4705 - (-0.4705) = 0;	-0.0014;	0.0000;
2.	0.9398 - 0.9398 = 0;	-0.0887;	0.0079;
3.	-0.2445 - (-0.2445) = 0;	0.0347;	0.0012;
4.	-0.4246 - (-0.4246) = 0;	-0.0286;	0.0008;
5.	0.9448 - 0.9448 = 0;	-0.0766;	0.0059;
50.	0.2152 - 0.2152 = 0;	0.0279;	0.0008;
51.	0.1826 - 0.1826 = 0;	-0.0411;	0.0017;
96.	0.6748 - 0.6748 = 0;	-0.0340;	0.0012;
97.	-0.7325 - (-0.7325) = 0;	-0.0044;	0.0000;
98.	0.3487 - 0.3487 = 0;	0.0082;	0.0001;
99.	0.2508 - 0.2508 = 0;	-0.0196;	0.0004;
100.	0.4771 - 0.4771 = 0;	-0.0306;	0.0009;

Table 7: **The best performing episode, with concrete values for steps** $1-5$**,** $50-51$**,** $96-100$**.** Symbolic expression used in this episode was $z = y - y = 0$.

6 Conclusion and Discussion

So far, RNNs, especially those employing LSTM units, have shown great successes. Through our experiments, we have shown one interesting application of recurrently connected SNNs and their LSTM-like units. Adaptive LIF neurons are endorsed with memory capabilities, and one can rely on this model to be able to store and retrieve information for longer timespans than for few hundred seconds, what was known before as a bottleneck when using SNNs with LIF neurons.

Through our first experiment, we have shown that information can be encoded and restored through identifiers, i.e., our network learned different nonlinear function, was able to restore the knowledge and apply it to new examples. This concept reminds of computational program with its execution routine, present in working memory and ready to be activated, when there is a demand for it. Once activated, inputs to this computational program are transformed according to execution routine, in our case, nonlinear function, providing very low errors.

Second experiment was similar, but more interesting, because network had to learn to use symbolic expressions, given as encoded rules, in fact, representing different functions, ranging from simple one, e.g., $z = 0$ to more complex one, $z = x * y$. Challenges here were:

- to learn to ignore one given variable and to use only another one (in expressions such as $x + x$, $y - y$, $x * x$...),
- subtraction is not a commutative function, so in this case it was very

important to use the given rule and arguments (concrete values of x and y) in the order given by the symbolic expression.

Another importance of this concrete experiment is for development of neuromorphic hardware. One could try to implement basic arithmetic operations with SNNs, which could possibly lead to more intelligent machines. If we think of us, humans, and our intelligence, without any doubt learning basic arithmetic is one of the first challenges we tacke in the school.

In our experiments we did not consider the arithmetic operation *division*, because of the mathematical constraint that division by zero is undefined, and our input values were centered around zero.

The goal of our third experiment was to test memory capabilities of our network and to see if the network was able to learn to use and remember the symbolic rule presented in the beginning of the episode, in as less steps as possible, and to apply that symbolic rule later even when it is not given. This experiment, compared to the second one (Learning symbolic expressions), suggested that the task was more challenging, but achievable, and it required a slightly more powerful architecture. Here it was essential to have greater memory capabilities (longer memory) of ALIF neurons, namely, τ adaptation time constant ($\tau_{a,j}$), whereas for the second experiment increasing it brought only insignificant improvements. Too "long" memory indicated that further enhancements of ALIF model are possible, for example, some kind of reset signal, or forget gate (analogously to LSTMs) would be useful for some purposes.

Appendices

A Network Parameters

Parameter	Value
number of recurrent neurons	250
proportion of adaptive neurons	0.4
dampening factor	0.4
tau adaptation	1-1000 ms
number of steps in an episode	500
duration of steps	20 ms

Table 8: **Network parameters for the experiment Learning nonlinear function using previous targets/function identifiers.**

Parameter	Value
number of recurrent neurons	300
proportion of adaptive neurons	0.4
dampening factor	0.3
tau adaptation	1-8000 ms
number of steps in an episode	100
duration of steps	40 ms

Table 9: **Network parameters for the experiment Learning symbolic expressions.**

Parameter	Value
number of recurrent neurons	350
proportion of adaptive neurons	0.4
dampening factor	0.3
tau adaptation	1-8000 ms
number of steps in an episode	100
duration of steps	40 ms

Table 10: **Network parameters for the experiment Remembering symbolic expressions.**

B References

[1] D. Brown. *Digital fortress*. St. Martin's Paperbacks, 2004.

[2] Jeff Hawkins and Sandra Blakeslee. *On Intelligence*.

[3] W. S. McCulloch and W. H. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, vol.5:115–133, 1943.

[4] Brenden M. Lake, Tomer D. Ullman, Joshua B. Tenenbaum, and Samuel J. Gershman. Building machines that learn and think like people. *CoRR*, abs/1604.00289, 2016. `http://arxiv.org/abs/1604.00289`.

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[6] W. Maass, C. H. Papadimitriou, S. Vempala, and R. Legenstein. Brain computation: A computer science perspective. *Draft of an invited contribution to Springer Lecture Notes in Computer Science*, vol. 10000, 2017.

[7] G. F. Marcus. *The Algebraic Mind - Integrating connectionism and cognitive science*. MIT Press, 2001.

[8] G. Marcus, A. Marblestone, and T. Dean. The atoms of neural computation - does the brain depend on a set of elementary, reusable computations? *Science*, 346(6209):551–552, 2014. incl. frequently asked questions.

[9] Christoph von der Malsburg. Binding in models of perception and brain function. *Current Opinion in Neurobiology*, 5:520–28, 1995.

[10] Chris Eliasmith, Terrence Stewart, Xuan Choo, Trevor Bekolay, Travis Dewolf, Charlie Tang, and Daniel Rasmussen. A large-scale model of the functioning brain. *Science (New York, N.Y.)*, 338:1202–5, 11 2012.

[11] Trenton Kriete, David C. Noelle, Jonathan D. Cohen, and Randall C. O'Reilly. Indirection and symbol-like processing in the prefrontal cortex and basal ganglia. *Proceedings of the National Academy of Sciences*, 110(41):16390–16395, 2013.

[12] Wolfgang Maass. wetware. *In TAKEOVER: Who is Doing the Art of Tomorrow (Ars Electronica 2001)*, pages 148–152, 2001.

[13] Wolfgang Maass. Networks of spiking neurons: The third generation of neural network models. *Neural Networks*, 10(9):1659 – 1671, 1997.

[14] Wolfgang Maass. Liquid state machines: Motivation, theory, and applications. 01 2010.

[15] W. Maass. Searching for principles of brain computation. *Current Opinion in Behavioral Sciences (Special Issue on Computational Modelling)*, 11:81–92, 2016.

[16] Guillaume Bellec, Darjan Salaj, Anand Subramoney, Robert A. Legenstein, and Wolfgang Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. *CoRR*, abs/1803.09574v2, 2018.

[17] W. Maass. To spike or not to spike: That is the question. *Proceedings of the IEEE*, 103(12):2219–2224, 2015.

[18] M. Davies, N. Srinivasa, T. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y. Weng, A. Wild, Y. Yang, and H. Wang. Loihi: A neuromorphic manycore processor with on-chip learning. *IEEE Micro*, 38(1):82–99, January/February 2018.

[19] Steve B. Furber, David R. Lester, Luis A. Plana, Jim D. Garside, Eustace Painkras, Steve Temple, and Andrew D. Brown. Overview of the spinnaker system architecture. *IEEE Trans. Comput.*, 62(12):2454–2467, December 2013.

[20] Johannes Schemmel, Daniel Brüderle, Andreas Grübl, Matthias Hock, Karlheinz Meier, and Sebastian Millner. A wafer-scale neuromorphic hardware system for large-scale neural modeling. In *ISCAS*, pages 1947–1950. IEEE, 2010.

[21] Steven K. Esser, Paul A. Merolla, John V. Arthur, Andrew S. Cassidy, Rathinakumar Appuswamy, Alexander Andreopoulos, David J. Berg, Jeffrey L. McKinstry, Timothy Melano, Davis R. Barch, Carmelo di Nolfo, Pallab Datta, Arnon Amir, Brian Taba, Myron D. Flickner, and Dharmendra S. Modha. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the National Academy of Sciences*, 113(41):11441–11446, 2016.

[22] Wolfgang Maass. Energy-efficient neural network chips approach human recognition capabilities. *Proceedings of the National Academy of Sciences*, 2016.

[23] Tom M. Mitchell. *Machine Learning*. McGraw-Hill Science/Engineering/Math, 1997.

[24] Jilles Vreeken. Spiking neural networks, an introduction. Technical report, 2003.

[25] Wulfram Gerstner and Werner Kistler. *Spiking Neuron Models: An Introduction*. Cambridge University Press, New York, NY, USA, 2002.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[27] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, vol. 521:436–444, 2015.

[28] Guillaume Bellec, Darjan Salaj, Anand Subramoney, Robert A. Legenstein, and Wolfgang Maass. Long short-term memory and learning-to-learn in networks of spiking neurons. *CoRR*, abs/1803.09574v1, 2018.

[29] Sepp Hochreiter, Arthur Younger, and Peter R. Conwell. Learning to learn using gradient descent. pages 87–94, 09 2001.

[30] Wikipedia. Cognition. [Online; accessed 01-December-2018].

[31] Dale Purves, Elizabeth M. Brannon, Roberto Cabeza, Scott A. Huettel, Kevin S. LaBar, Michael L. Platt, and Marty Woldorff. *Principles of Cognitive Neuroscience*. Sinauer, 2008.

[32] Patricia S. Goldman-Rakic. Working memory and the mind. *Scientific American*, 267:110–7, 10 1992.

[33] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014.

[34] Adam Santoro, Sergey Bartunov, Matthew Botvinick, Daan Wierstra, and Timothy P. Lillicrap. One-shot learning with memory-augmented neural networks. *CoRR*, abs/1605.06065, 2016.

[35] Kenneth Hayworth. Dynamically partitionable autoassociative networks as a solution to the neural binding problem. *Frontiers in Computational Neuroscience*, 6:73, 2012.

[36] T. R. Besold and K.-U. Kühnberger. Towards integrated neuralsymbolic systems for human-level AI: Two research programs helping to bridge the gaps. *Biologically Inspired Cognitive Architectures*, 14:97–110, 2015.

[37] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.