SIMULATION AND GENETIC EVOLUTION OF SPIKING NEURAL NETWORKS

Petr Podhorský, Miroslav Skrbek

Czech Technical University in Prague, Faculty of Information Technology,
Department of Computer Systems,
Computational Intelligence Group,
Kolejní 550/2, 160 00 Praha 6, Czech Republic

podhope1@fel.cvut.cz (Petr Podhorský)

Abstract

In this paper, an implementation of a simulator for spiking neural networks and learning algorithm using genetic evolution is described. We have implemented two neural models (simplified Spike Response Model and Integrate and Fire model) and two learning algorithms – SpikeProp (its original version modifying only weights and an enhancement for changing all network variables – weights, delays, time constants and thresholds) and a simple genetic learning algorithm (presented in this paper). This learning algorithm is easy to understand and does not require any special network topology like feed-forward networks and back-propagation algorithms or thorough investigation of network architecture. It is based on an assumption that small changes in network variables have a small impact on the output and big changes have a big impact, thus calculating a difference between a desired output and a real output and mutating individuals according to a size of this difference we can expect a population to converge. We verified this approach on frequently used benchmarks.

Keywords: Eurosim, congress, spiking neural networks, genetics, evolution.

Presenting Author's biography

Petr Podhorský is a Ph.D. student at Czech Technical University in Prague, Faculty of Information Technology, Department of Computer Systems. After getting his master degree in 2009, he has joined the Computational Intelligence Group. His research focuses on spiking neural networks.

1 Introduction

Spiking neural networks [5] are labeled as third generation networks. A reason for this is a biological more realistic model, where time factor is involved. Simulation of spiking neural networks is more complex than with classic neural networks due to a higher count of network variables and more complex timing. Because of a failure to find a proper simulator for educational and experimental purposes, we decided to create our own simulator for studying spiking networks behavior and properties.

In this paper, we present our spiking neural network simulator containing our genetic algorithm for learning spiking networks involving an adaptive mutation. The simulator supports simplified Spike Response Model and Integrate-and-fire model.

Along the implementation of the simulator we wondered if it is possible to solve the learning problem for spiking neural networks with genetic algorithms (instead of using some kind of back-propagation technique). This was due to the fact that we are convinced that genetic algorithms are more powerful for problems with many variables than gradient algorithms (like back-propagation).

Overview of spiking neural networks is in Section 2. It describes neuron models as they were implemented in the simulator.

Section 3 describes SpikeProp (back-propagation based algorithm), which was used for comparison to our genetic algorithm in presented experiments.

Section 4 describes our proposed genetic algorithm in detail. The simulator which implements this algorithm is presented in Section 5. Simulation results are described in Section 6.

2 Spiking neural networks

A main difference between classic ANNs and SNNs is that values are not carried directly along connections, but by spikes propagated along synapses between neurons. It allows an arbitrary spatiotemporal coding, for example rate coding, linear temporal coding, population coding, rank order coding etc.

There are several spiking neuron models differing in complexity and a mathematical description. Models we used in the simulator are SRM (Spike Response Model) and Integrate-and-fire model.

2.1 Spike Response Model

SRM implemented in simulator describes an internal potential of a neuron like:

$$u(t) = \sum_{s=0}^{N} w_s \sum_{p=0}^{N_s} \frac{t - t_p}{\tau_s} e^{1-\frac{t-t_p}{\tau_s}} \qquad (1)$$

First sum goes over all input synapses to the neuron, second one stands for all spikes coming in across the synapse. u(t) represents the neuron potential, w_s a weight of a synapse, t a current time and t_p a time when spike arrived to the neuron, τ_s a time constant for a particular synapse.

A simulation of a network with SRM model is made in a clock-driven fashion, i.e. a network is simulated in small time steps.

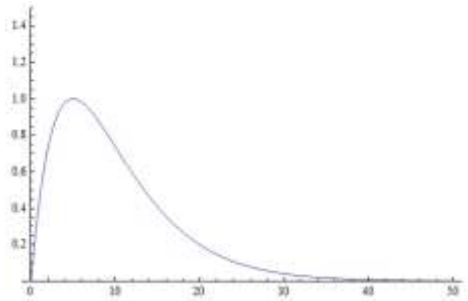

Fig. 1 Spike Response Model (an effect of a spike to a neuron's potential)

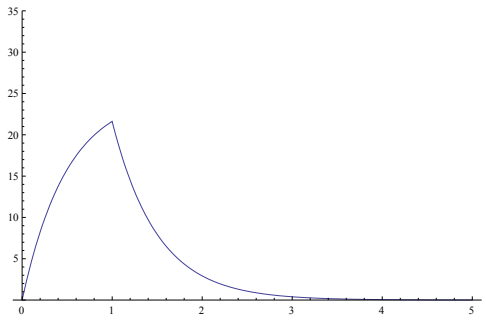

Fig. 2 Integrate-and-fire model (an effect of a spike to a neuron's potential)

These figures depict shapes for both models. A concrete shape and values are related to constants, so these figures are only illustrational.

2.2 Integrate-and-fire model

The model of spiking neuron is more complex than SRM model. It can be simulated in an event-driven fashion, thus it runs faster than clock-driven simulations.

$$\tau * V' = V_{Rest} - V + R * J \qquad (2)$$

Differential equation (2) describes the Integrate-and-fire model [4]. V represents an internal neuron's potential, V_{Rest} a resting potential, J a synaptic current, R a neuron's resistance and τ a neuron's time constant. Different variable names than in SRM are used because of a different literature source of this neuron model.

The synaptic current J is calculated as a sum of rectangular shaped pulses (with a synapse's weight as a height of this pulse).

3 SpikeProp

SpikeProp is a back-propagation learning mechanism derived for spiking neural networks introduced in [1].

In classic ANNs, an error of the network is calculated as a difference between an actual and a desired output. For SNNs, back-propagation algorithm was derived for first time spike coding (e.g. linear temporal coding) and the error is calculated as a difference between an actual and a desired output spike time.

$$\mathrm{E} = \frac{1}{2}\sum_{j \in O}(t_j^a - t_j^d)^2 \qquad (3)$$

In each iteration parameters (weights only or all neuron'/synapses' parameters) are modified by the delta rule as follows:

$$\Delta \mathrm{p_i} = -\eta_\mathrm{p}\frac{\partial E}{\partial p_i}, \qquad (4)$$

where Δp_i is a change of a i-th parameter, η_p is a learning rate for specific parameter class and $\partial E/\partial p_i$ is a partial derivative of E by parameter p_i.

SpikeProp iterates until the error falls below predefined threshold, a maximum number of iterations is reached or some neuron stops firing (in this case it is not possible to calculate the error).

Because of nature spiking networks, some problems, unseen in classical ANNs, can pop up – e.g. because weights are being altered by difference between output spike times and desired times, one cannot compute this difference anymore, when neuron stops firing.

4 Genetic learning algorithm

A genetic algorithm (GA) is a strong optimization method. It involves similar processes known from nature – evolution of a population, selection of best individuals and their mutation or crossover. An advantage of this approach is a lack of need to study a concrete problem thoroughly – e.g. for a feed-forward network, there is no need to study and to implement some kind of a back-propagation technique.

The proposed genetic algorithm evolves a population of individuals. Each individual is represented by one spiking neural network (i.e. there is no conversion to binary format whatsoever). In each iteration, worst individuals which are removed from population are replaced by mutated survivors. Generation of new individuals is made by mutation of parameters, no crossover is performed. Individuals are evaluated according to an error of the network, calculated as follows:

$$\mathrm{E} = \frac{1}{2}\sum_{p \in P}(y_p^a - y_p^d)^2, \qquad (5)$$

where E is the error, P is a set of patterns presented to the network (e.g. 4 patterns for the XOR problem), y_p^a is a real network output and y_p^d is a desired output.

A key assumption for our algorithm is that it's more plausible to make greater changes to individuals that are farther from the ideal solution – a random value from range [-0.5;0.5] is multiplied by an error of the individual and some constant specific to the parameter. These constants were empirically set based by a few experiments (if the constant is a zero, the parameter remains unchanged, if it is too high, only bad individuals are produced, a good value is somewhere inside this interval). Constants used were 10^5 for weights, 10^2 for delays, 10 for time constants and 10^2 for neurons' thresholds.

A following equation represents an operation for each parameter (p) for each individual in the population. A rnd function generates a random number in the interval 0 and 1 and P_p represents a probability that the parameter will be mutated.

$$\Delta \mathrm{p_i} = \begin{cases} \eta_p \mathrm{E}(\mathrm{rnd}() - 0.5), & rnd() < \mathrm{P_p} \\ 0, & otherwise \end{cases} \qquad (6)$$

A fitness value can be expressed as a negative error, so an individual with the best fitness will have a zero error value.

A pseudo-code illustrating how the implemented genetic algorithm works follows.

1. Initialize initial population
2. Calculate fitness values for all individuals
3. Remove N worst individuals
4. Generate N new individuals by duplicating and mutating survivors
5. Calculate fitness values for all individuals
6. If the error of the best individual didn't fall below some threshold, go to step 3.
7. Take the best individual as a solution

Fig. 7 Pseudo-code illustrating the implemented genetic algorithm

We can also look at this learning algorithm as at some genetic variation of a gradient descending algorithm. In the first step, a population is initialized somewhere in the variable space. Better individuals survive to a next population, worse ones are replaced. Because each offspring is generated from some relative good individual (a parent is uniformly selected from N best individuals in the population) and a size of a mutation is related to an error, population moves closer and closer to a local/global minimum.

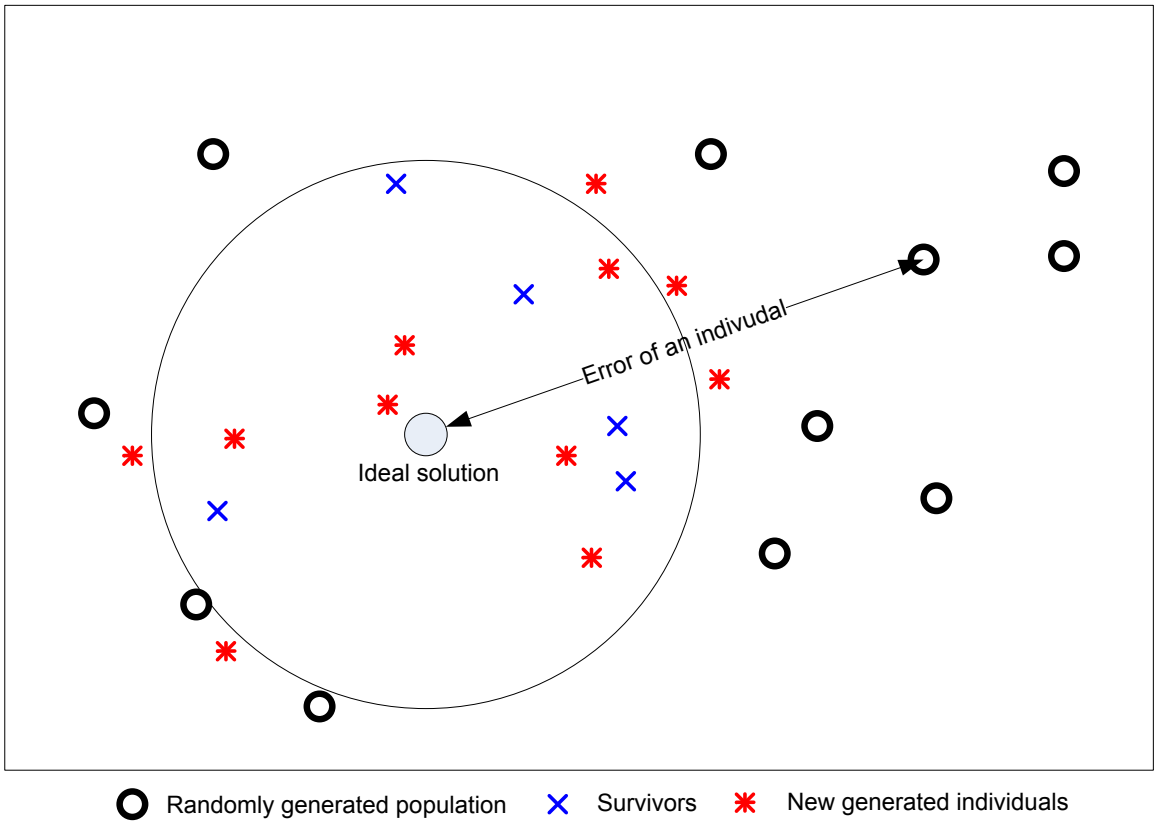

Fig. 3 Genetic evolution in the variable space. This picture illustrates initializing a first generation (black circles), selecting the best individuals (blue crosses) and creating new individuals (red stars) closer to the optimal solution.

5 Simulator

We implemented a simulator for educational and experimental purposes, because we didn't find proper one for our needs. Although we found pure simulators for spiking neural networks, we didn't find some easy-to-use and sufficiently simple simulator with learning algorithms suitable for spiking neural network exploration and spiking activity visualization.

Our simulator implements Spike Response Model and Integrate-and-fire model (see Section 2) and provides visualization of running spikes and changing neuron potentials in time. It is based on Java platform because of portability and because Java is well spread.

5.1 Simulation types

The simulator implements two known approaches for simulation – a clock-driven and an event-driven. The clock-driven simulation calculates all network variables at each step of the simulation and checks whether something important happened. This logic is easy to understand on the one hand on the other hand it involves a lot of computations. Simplified Spike Response Model is computed in our simulator in this fashion, i.e. at each time step all neuron potentials are computed and checked for crossing thresholds. If any neuron's potential crosses its threshold, it is set to zero and spikes are sent to output synapses of that neuron.

The event-driven approach is more difficult than clock-driven, but brings some advantages. Because of its nature, only important events in the network are simulated thus unnecessary computations are eliminated. Therefore, computation complexity is related to what happens in the network (how many spikes are emitted and how often) in opposite to clock-driven simulation where complexity is related to a structure of the network and duration of a simulation (for each time step, all neuron potentials are computed). We use this approach for Integrate-and-fire model, because it's described using a differential equation, which can be used for determining if and when a neuron's potential crosses a threshold.

5.2 Simulator interface

The simulator has a graphical interface for visualizing network architectures and activity (running spikes) in a simulated network for educational purposes.

Fig. 4 shows a graphical user interface of the simulator with a network for the XOR problem depicted as an example.

Using the graphical interface user can create his/her own experiments – besides predefined experiments

thoroughly described later. This option is currently

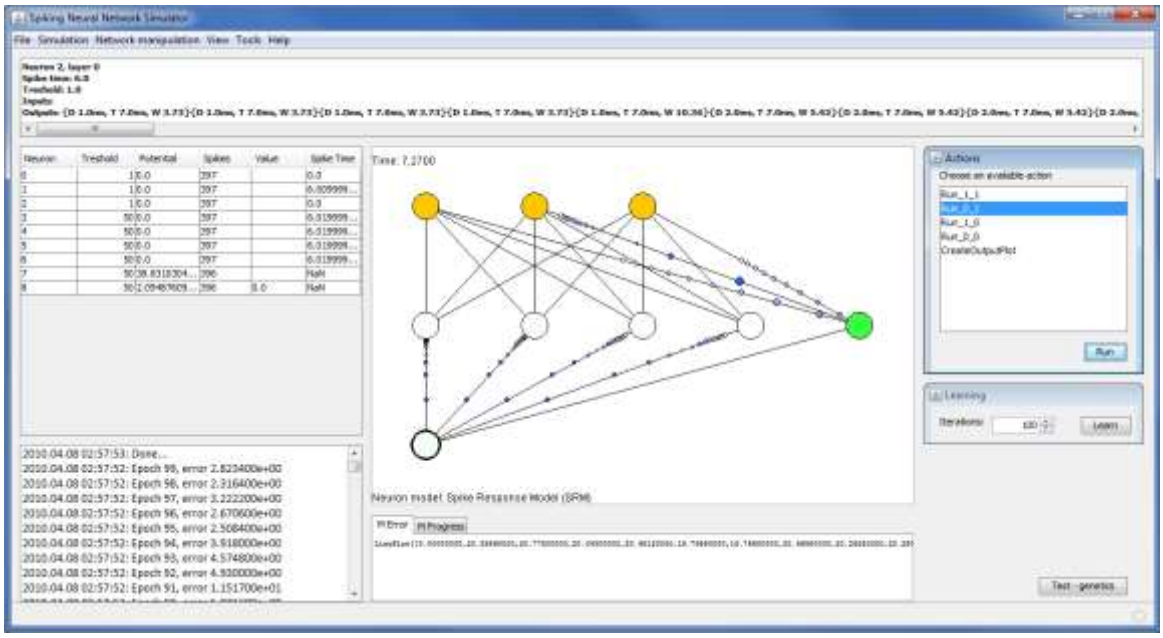

Fig. 4 User interface of our spiking neural network simulator (a network with 3-5-1 architecture for the XOR problem depicted)

under development, but so far, it is possible to select from an arbitrary network (n neurons is placed and user manually creates synapses between them) or a feed-forward network (whole network is generated according to input parameters). Input and output data are read from a XML file. Currently, only encoding with first spike times is supported. A problem with SNNs is that network can work with any possible encoding on its input and output – spikes could be interpreted in any way. The most suitable solution for this would be probably a special algorithm for each problem, but we focus on the basics so far.

6 Experiments

Simulator has been tested on standard XOR benchmark and the Iris [6]. Architectures and data for these problems are predefined, so user only selects an experiment and runs it.

6.1 XOR problem

A first experiment was to train a network for a XOR problem (because it's a simple, but non-linearly separable classification problem) as it was made in [1]. Architecture is 3-5-1 (number of neurons in input, hidden and output layer) with 16 synapses between each pair of neurons. Each synapse has a fixed delay varying from 1 to 16 milliseconds. A reason for this architecture is a fact that only weights are trained, not delays or time constants, and there have to be a way how the learning process can "pick" a proper delay (although one can expect, according to results, the trained weight vector does not look like "1 from N").

A next step in this experiment involves learning all synapses' and neuron' parameters, i.e. delays, time constants, weights and thresholds, as it was made in [2]. An advantage of this approach is a less complex structure and therefore faster simulation.

Architecture for this test is also 3-5-1, but there are only two synapses between each pair of neurons.

The XOR input patterns were encoded using linear temporal coding, coding 1 at t=0 ms and 0 at t=6 ms. One spike was always sent at t=0 as a reference spike. Decoding a network's output is similar – 1 is represented by an output spike time at t=10 and 0 at t=16.

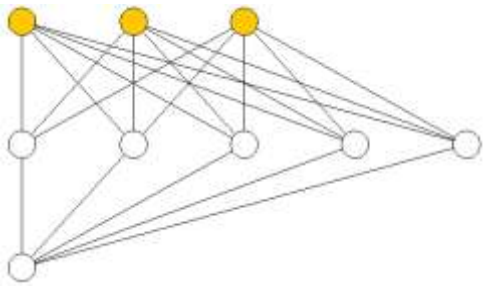

Fig. 5 Network architecture for the XOR problem (3 input neurons, 5 hidden neurons, 1 output neuron)

Networks were trained until an error fell under 2.0 milliseconds. The error for a network was calculated according to the equation (3).

For the architecture with 16 synapses between each pair of neurons, several weights initializations for back-propagation were tried and the best one was to initialize all weights to a value of 6.0 (neurons' thresholds 50.0). Other initializations were: different constant values, random values between 1 and 10 and

random values between 1 and 10, but always same weights for one pair of neurons.

Tab. 1 Different weight initialization methods for XOR

Weight initialization method	**XOR 3-5-1, 16 synapses, back-propagation**
Constant value – 4	144 for 100,00%
Constant value – 5	117 for 100,00%
Constant value – 6	**103 for 100,00%**
Constant value – 7	144 for 100,00%
Constant value – 8	186 for 100,00%
Random value between 1 and 10, but all same for one connection	198 for 92,00%
Random value between 1 and 10	131 for 100,00%

For the second architecture with two synapses per one connection, constant value of 0.5 was used (neurons' threshold was 1.0).

In the table (Tab. 2) are written down results for this problem. Each combination (structure/learning method) was observed for two values – how many epochs were needed for the error to fall under a specified value and also how many configurations (different set of weights in a starting configuration or different path of evolution) the network converged for.

Tab. 2 Results for the XOR problem

	Back-propagation	**Genetics (SRM)**	**Genetics (IF)**
XOR (3-5-1, 16 synapses)	103 for 100 %	189 for 60,00%	197 for 70 %
XOR (3-5-1, 2 synapses)	109 for 88,00%	125 for 100,00%	92 for 90,00%

The results showed that a genetic algorithm is a suitable way for training a network on a simple classification non-linearly separable problem (independently on a neuron model).

6.2 Iris benchmark

The Iris dataset consists of 150 cases divided into three classes (50 instances for each class) which are not linearly separable. There are 4 input variables describing each instance and its correct classification. These input variables were encoded by 12 neurons for each one of them (thus 48 neurons in total) using population encoding described in [1].

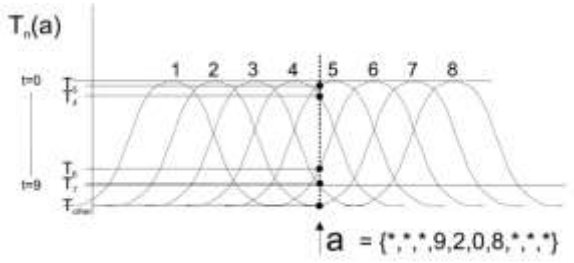

Fig. 6 Population coding illustration - we are encoding value "a" by 8 neurons (figure taken from [1]).

Population coding is used for a greater resolution than in a simple linear temporal coding. This type of coding is more biological plausible, because neurons can distinguish spikes only to certain limits (according to [1], cortical times are inherently noisy and a resolution under 1-2ms is not possible). Using more neurons to encode one value it is possible to resolve values under this resolution.

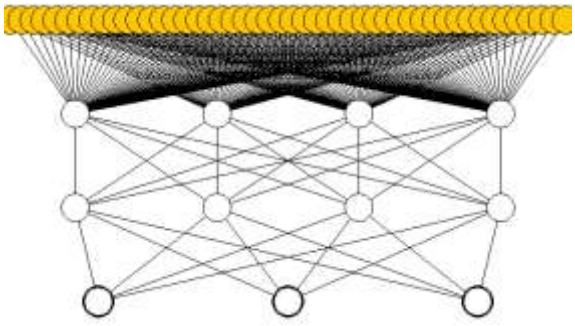

Fig. 7 Network architecture for the Iris problem (8 hidden neurons in two layers, 3 output neurons for 1:N coding)

For this benchmark, the experiment was ran 10 times and always learned for 300 epochs. Data was separated into two sets, training (70% from the original set) and testing (30%).

Output encoding for this benchmark was 1:N, i.e. first firing output neuron is a winner and points out a predicted class. The error is calculated as a square of difference between an ideal firing time and actual firing time, where the ideal firing time is a firing time of a neuron representing a correct class. Additionally, the difference has to be bigger than some minimum value (because of noise in synapses and neurons).

Tab. 3 Results for the Iris problem

Results after 300 epochs	Training set	Testing set
Test 1	96,23%	90,91%
Test 2	98,10%	91,11%
Test 3	95,10%	79,17%
Test 4	97,32%	92,11%

Test 5	92,52%	86,05%
Test 6	97,09%	97,87%
Test 7	92,08%	95,92%
Test 8	97,22%	97,62%
Test 9	94,00%	88,00%
Test 10	97,27%	87,50%

According to results, the network provided good prediction and proved it can be learned for this type of problem with a population coding using our genetic algorithm.

7 Conclusions

We have successfully created a simulator for experimental purposes and presented a simple genetic learning algorithm for spiking neural networks. The simulator allows running experiments, displaying network architecture and studying spiking networks. For learning we have implemented SpikeProp [1] with enhancements [2] and the genetic algorithm presented in this paper.

The proposed genetic algorithm has been tested on standard benchmark tests and it showed we were capable to learn spiking neural network for these datasets. Experiments showed we can use this approach with both used neuron models (SRM and IF) and without any specific knowledge of the network structure.

8 Acknowledgment

This work has been supported by the research program "Transdisciplinary Research in the Area of Biomedical Engineering II" (MSM6840770012) sponsored by the Ministry of Education, Youth and Sports of the Czech Republic.

9 References

[1] Bohte, S. M., Kok, J. N., and Poutre, J. A. L. Error-backpropagation in temporally encoded networks of spiking neurons. Neurocomputing, 48(1-4):17—37, 2002.

[2] Benjamin Schrauwen, Jan Van Campenhout. Improving SpikeProp: Enhancements to An Error-Backpropagation Rule for Spiking Neural Networks. Proceedings of the 15th ProRISC Workshop, 2004.

[3] N.G. Pavlidis, D.K. Tasoulis, V.P. Plagianakos, G. Nikiforidis and M.N. Vrahatis. Spiking Neural Network Training Using Evolutionary Algorithms. Intemational Joint Conference on Neural Networks, Montreal, Canada, July 31 - August 4, 2005.

[4] Romain Brette, Michelle Rudolph, Ted Carnevale and others. Simulation of networks of spiking neurons: A review of tools and strategies, February 4, 2008.

[5] Wolfgang Maass. Networks of Spiking Neurons: The Third Generation of Neural Network Models. Neural Networks, Vol. 10, No. 9, pp. 1659-1671, 1997.

[6] UCI Machine Learning Repository: Iris Data Set. http://archive.ics.uci.edu/ml/datasets/Iris.