|  |  |
| --- | --- |
|  | ArNI-X: A simulator for spiking neural networks |
|  |  |
| 12/20/2023 | Version 2.0 |
|  | By Mikhail Kiselev |

ArNI-X: A simulator for spiking neural networks

Version 2.0

# CONTENTS

Copyright.....................................……………………………………………………....... iv

Introduction……………………………………………………………………………………. 1

General information....................................................................................................... 1

Системные требования .......................................................................................... 2

Установка пакета……….......................................................................................... 3

Модели нейронов и синапсов ..…………………………………………………………. 4

Описание структуры сети ……...…………………………………………………………. 7

Эмуляция входного сигнала ИНС ………………………………………………………. 10

Интерфейс генераторов входного сигнала ......................................................... 10

Конфигурирование ИНС …………..………………………………………………………. 12

Интерфейс динамической библиотеки - конфигуратора ИНС............................. 12

Интерфейс конфигуратора ИНС INetworkConfigurator………............................... 13

Реакция на активность выходной популяции ИНС…..……………………………. 14

Интерфейс интерпретатора выходного сигнала……………............................... 14

Эмуляция ИНС…………………………………………………………………………………. 15

Общая логическая структура компонент, отвечающих за эмуляцию ИНС ...... 15

Командная строка запуска эмуляции ИНС ……………......................................... 15

Результаты эмуляции……………….……………………......................................... 16

GPU реализация – проект ArNIGPU …..…………………......................................... 17

CPU реализация – проект ArNICPU ..……………………......................................... 20

Обработка протокола активности сети ………………………………………………. 21

Визуализация мониторинга сети – Results …………………………………………. . 21

Оптимизация свойств ИНС с помощью генетического алгоритма – проект ArNI\_GA ……………………………………………………………………………..…………. 22

Оптимизация свойств ИНС с помощью координатного спуска – проект ArNI\_CD…………………………………………………………………………………………. 24

Структурный генетический алгоритм - проект ArNIGAstruct ……..…………… 25

Процедура верификации модели нейрона………………………………..…………. 26

# Copyright

Copyright © 2007 – 2023 Mikhail V. Kiselev

Mikhail Kiselev is author and owner of ArNI-X.

The third party packages used:

1. Boost C++ library (license <https://www.boost.org/users/license.html>).
2. Pugixml library (<http://pugixml.org>) under the MIT license. Copyright (C) 2006-2018 Arseny Kapoulkine

# Introduction

## General Information

The ArNI-X system is used to simulate spiking neural networks (SNN) on CPU and GPU. For sake of maximum performance, it is written in the C++ and CUDA programming languages. At present, these exist versions for Windows and Linux. There are 3 modes to use ArNI-X corresponding to different trade-offs between flexibility and ease of use:

* **XML mode.** **In this regime, any programming is not needed.** The network structure is defined in a special XML-based declarative language using one of the built-in models of neurons and synaptic plasticity and combining various standard neuron connectivity patterns. The present manual describes only this mode of ArNI-X usage.
* **API mode.** In case of custom neuronal structures, non-standard input spike sources and network activity post-processors, it is possible to implement all these features using the C++ API provided. The neuron and synaptic plasticity models are standard.
* **Source modification mode.** If the emulated neuron model does not fit the model class implemented in ArNI-X, the source modification will be required. The ArNI-X code is written so that to make the process of new model implementation as easy as possible but discussion of the respective techniques is beyond the scope of this manual.

The implemented neuron models are rather simple and are hardly suitable for detailed neurophysiological modelling. ArNI-X is more oriented to practical applications, creation of prototypes of SNN-based devices solving real-world problems. A separate important goal is prototyping possible implementation of SNNs on modern (e.g. Inlel’s Loihi) and future neuroprocessors. Keeping it in mind, we tried to make the models of neurons and synaptic plasticity as simple and hardware-friendly as possible.

In order to illustrate how to emulate SNN using ArNI-X, we include a simple example at the end of this manual. The reader of this manual is assumed to have basic knowledge of SNN theory. More non-standard or advanced concepts are explained as they appear in the text.

## Installation

## Running SNN Emulation using ArNI-X

ArNI-X executable modules are realized as console applications. They are launched in a directory that will be referred to as working directory. All emulation results are saved in this directory. It is assumed that computational experiments with SNNs go in series such that every individual emulation run has the series name and the numeric id in the series. In order to run a simulation, the user should describe SNN structure using a special file <experiment\_numeric\_id>.nnc. All nnc files related to one experiment series should be in one separate directory. The nnc files contain SNN structure definition in XML language; the format of these definitions is described in the subsequent sections of this manual. The emulation is performed by executable files ArNICPU (for CPU) or ArNIGPU (for GPU). The experiment series name and the experiment id are specified in command line arguments (see below).

The simulation duration is specified in the command line, as well.

The GPU version works with NVIDIA GPUs with compute capability at least 5.2.

# Tutorials

## Single Neuron

The network configuration (.nnc) files used in the Tutorials are in the sub-directory Tutorials of the ArNI-X root directory. Tutorial #1 illustrates a network consisting of a single neuron. It is described in the file 1.nnc. You can change the neuron parameters and see how its activity changes. The Tutorials works on any computer even without GPU and therefore are based on the CPU version of the emulator ArNICPU. We recommend running the emulator from the Workplace sub-directory. It can be done by the command line

ArNICPU ..\Tutorials -e1 -Pt

In the Linux systems, this line should be prefixed by ./ and slash should be used instead of back slash. This command line means that the configuration file ..\Tutorials\1.nnc will be used and the text file containing network activity record will be created (-Pt).

Text network activity protocol is the simplest form of recording network activity. Every its row corresponds to one emulation step, every column corresponds to a neuron. If the given neuron fired on the given step, it will be denoted by the character ‘@’ in the respective position. Otherwise, the character would be ‘.’.

The nnc files are written in XML language. The top level node is always SNN.

The input spike sources are defined in nodes RECEPTORS. Every receptor section should have a name (here, it is R) and input node count (10 – in our case). Input spike sources are implemented as dynamic libraries. Details of the implementation are described in the node Implementation. The present manual covers only one input spike source type – fromFile. This source reads input spikes from file and adds Poissonian noise to them. If the file name is not specified, only Poissonian noise is sent to the network. The noise node inside args node defines the noise intensity. Namely, this number *f* is probability of input spike from one node in one emulation step. Thus, the mean noise frequency for each input network node is 1000*f* Hz. The history\_length node specifies the emulation duration. It is equal to 1000. As in many SNN studies, we assume that one emulation step corresponds to 1 msec, so that our emulation will be 1 sec long.

The network itself is described inside the NETWORK node, particularly, in the Sections node. There may be other nodes inside NETWORK except Sections, but they are used to describe network parts implemented by separate dynamic libraries – this feature is not covered by the present manual. The Sections node includes two types of nodes – Section and Link. The former describes various neuron populations (or network sections), the latter – connections between populations (also called projections).

Our example contains only one section consisting of one neuron. Its name is neuron. The section properties are defined in the props node. In our case, this single neuron is the simplest leaky integrate-and-fire (LIF) neuron. This neuron has only one property – membrane potential decay time constant. Here, it equals to 10 msec (see the node chartime). Each Section node should contain the n node, which specifies the number of neurons belonging to this section.

Every Link section defines one projection type. Projection always connects neurons from two neuron sections or an input node section with a neuron section (projections from a neuron section to itself are also allowed). In our example, the only projection is from input nodes to our single neuron. The connection policy is all-to-all. Connections (projections) also have some properties. The most important one is the synaptic weight. In our neuron model, it is a value by which the membrane potential changes when the synapse receives a spike.

When the membrane potential reaches the threshold value *H*, the neuron fires and the membrane potential is reset to 0. In the LIF neuron model implemented in ArNI-X *H* = **8.531**. Such a strange value is explained by historical reasons. On the early stages of our research project, we experimented with analog neurons implemented in hardware with fixed threshold potential equal to 0.8531 Volts. This constant then migrated to numerous and diverse software models so that even after end of our hardware experiments we decided not to change this value to, say, 1 because it would require too many changes in many places. If threshold equal to 1 or to 0.02 Volts (the difference between the threshold and rest potentials in living neurons) seems more preferable, it is easy to get just by multiplying all synaptic weights by the respective constant.

After you execute emulation using the command line above, you see the new file spikes.1.txt. This file contains only one column showing activity of the neuron. Counting number of the ‘@’ characters we obtain the mean neuron’s firing frequency equal to 259 Hz. Varying the input synaptic weight, you can see how the neuron activity changes.

## Liquid State Machine

In the previous tutorial, we worked with a single neuron. The present tutorial is devoted to exploration of behavior of many interconnected neurons. A large ensemble of chaotically interconnected neurons can be used for classification of spatio-temporal patterns. The idea of this classifier (it is called Liquid State Machine) is the following. Changing in time streams of input spikes which reflect dynamics of a certain process are sent to input nodes of the chaotic SNN. This stimulation induces network activity. Since the SNN is recurrent, it has memory in the sense that its current activity depends on current stimulation as well as on stimulation in more or less distant past. Signals travelling in the SNN keep information about recent input spikes. Current network activity measured in terms of the mean firing frequencies of its neurons is different for different dynamics of input spike streams in the recent past, and therefore, it can be used by an external classifier for recognition of spatio-temporal patterns. Of course, this mechanism works only in the networks with certain characteristics of their neurons and connectivity. It makes exploration of properties of such chaotic networks important.

In this tutorial, the chaotic SNN described in the file 2.nnc includes two neuron populations – excitatory and inhibitory. The synaptic weights of connections from the former neurons are positive, from the later – negative. The names of these populations are E and I, correspondingly. Excitatory neurons are stimulated by Poisson noise – as in the previous tutorial. But these connections are not “all-to-all”. The projection property probability determines probability that the given input nodes is connected to the given neuron. There are 700 excitatory neurons and 300 inhibitory neurons in the network.

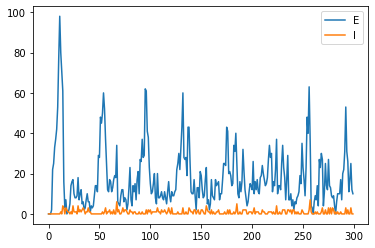
However, this time, excitatory neuron model is more complex than simple LIF. Threshold potential of these neurons is not constant. Every time the neuron fires it is incremented by 1 (see the threshold\_inc parameter); after that it linearly drops to its rest value **8.531**. The speed of this decrease is controlled by the parameter threshold\_decay\_period. Namely, every emulation step, threshold potential decrease by the value of threshold\_inc divided by the value of threshold\_decay\_period. This model is called LIFAT (leaky intergrate-and-fire neuron with adaptive threshold). This feature provides the network with the homeostatic property – it is hard for too active neurons to increase their activity more because of the high value of their threshold potential.

The file 2.nnc describes connections between these two kinds of neurons in four Link XML nodes. We see that all postsynaptic connections of excitatory and inhibitory neurons have identical properties. It should be noted that excitatory connection have another important property, in addition to synaptic weight. It is synaptic delay – number of emulation step necessary for transition of a spike from the presynaptic neuron to the postsynaptic neuron. By default, it is 1 but for excitatory connections in this example it is a random value from the range [1, 30]. We see also that inhibitory connections have the great negative weight.

Run the simulation by the command line

ArNICPU ..\Tutorials –e2 -Pt

Now let us look at the firing frequency dynamics for the E and I populations. It is drawn by the python script DrawSectionActivities.py if to run it in the Tutorials directory. Here it is:

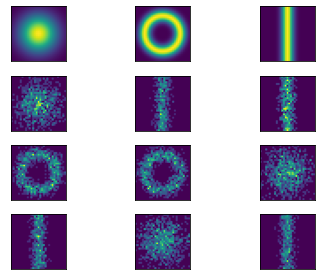


We see that the E population demonstrates non-trivial rhythmic behavior with the frequency about 20 Hz.

## Unsupervised Learning

In the previous tutorials, we considered behavior of networks which did not change during emulation. However, the most valuable property of neural networks is their ability to learn via the appropriate modification of their synaptic weights. This ability is demonstrated in the present tutorial for a case of unsupervised learning. In this learning regime, several different classes of stimuli are presented to the network many times. During the learning process, network’s synaptic weights should be adjusted in such a way that to form specific reaction of network to different stimulus classes – there should be neurons in the network which respond by elevated firing frequency to presentation of stimuli from one specific class.

In this tutorial, we put small monochrome images of 3 different classes onto the network’s input. Each image has size 31×31 pixel, therefore, the network has 961 input nodes – one input node corresponds to one pixel. Each image presentation lasts 10 msec. The probability that the given input node emits spike in one emulation step is proportional to the pixel brightness and is equal to 0.3 for the brightest pixels. Thus, the image presentations from the same class are not identical that makes the task non-trivial. In our example, the image classes are the spot in the center, the fuzzy ring, and the fuzzy vertical line. See the upper row of the picture below. The several examples of particular images from all these classes are depicted in the next 3 rows.



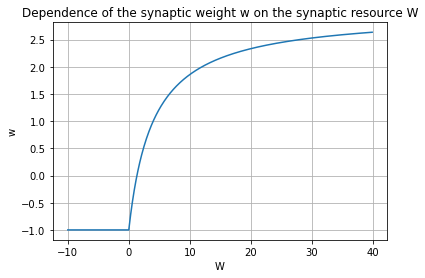
The file containing input spikes is clusters.txt (in the Workplace directory). Its format is the same as for network activity record. The respective cluster numbers (we will need them to evaluate the learning results) are contained in the file cluster\_labels.txt. The images are presented in random order, the total number of images is 3000.

This unsupervised learning problem will be solved by the simplest possible network described in the 3.nnc file. It consists of only 3 neurons. Respectively, we would like that each neuron reacted only to images from the class it learnt to recognize so that there should be a direct correspondence between the neurons and the image classes.

The neurons are connected to the input nodes but these are not “all-to-all” connections. In order to learn to recognize different patterns, the neurons should be sufficiently different from the very beginning. One way to get it is to connect neurons with different subsets of input nodes (but these subsets should be sufficiently large – otherwise the neurons would not obtain enough information about the input stimuli). The connection probability 0.9 seems to be a good trade off. In contrast with the previous examples, the connections with input nodes are plastic – their weights can change making learning possible. It is determined by their type – plastic. We told about importance to make the neurons different. Another way to do it is to provide them with different (and random) original weights.

Here, we should make an important remark. In our plasticity model, the plasticity rules are not applied to synaptic weights directly. Instead, the plasticity mechanisms modify another property of synapses called *synaptic resource*. The value of synaptic resource *W* depends monotonously on the synaptic weight *w* in accordance with the formula

In this model, the weight values lay inside the range [*wmin, wmax*) - while *W* runs from -∞ to +∞, *w* runs from *wmin* to *wmax* (see the picture below, where *wmin* = -1, *wmax* = 2).



Such an approach allows solving the important problem of catastrophic forgetting. Indeed, let us imagine that network was being trained to recognize something for a long time. As a result, the majority of synaptic weights became either saturated (equal to the maximum possible value) or suppressed. However, presentation of even few wrong training examples or examples containing other patterns or simply noise is sufficient to destroy the weight configuration learnt and nothing can prevent it. The network will forget everything it has learnt. But in our model, when *W* is either negative or highly positive, synaptic plasticity does not affect a synapse’s strength. Instead, it affects its stability – how many times the synapse should be potentiated or depressed to move it from the saturated state. Thus, to destroy the trained network state, it is necessary to present the number of “bad” examples close to the number of “good” examples used to train it. It should be noted that this feature was found to be useful for various learning tasks, not only for unsupervised learning.

Let us return to our network. We make input connections of the network different initializing them with random synaptic resource from the range (0, 2.9739) as it is specified by the XML node IniResource in the respective Link node.

The *wmin* to *wmax* values are set equal for all plastic synapses of a neuron. In our model, they are properties of neuron (strictly speaking, they are properties of a neuron population). They are set in the XML nodes minweight and maxweight, respectively.

Now it is time to tell about our synaptic plasticity rules. There are several plasticity mechanisms in our model. The first one is the so-called Hebbian plasticity. Donald Hebb’s law of synaptic plasticity states that the synapses, which helped the neuron to fire, are potentiated. Since effect of a presynaptic spike on membrane potential decreases with time constant defined in the chartime node, we can conclude that synapses obtaining last spike long time (sufficiently greater than chartime) ago before neuron firing do not contribute to it. In our model, Hebbian plasticity affects the synapses obtaining spikes in the period before firing equal to 3 \* chartime. Our version of Hebbian plasticity is very simple. Synaptic resource of every synapse having obtained a spike 3 \* chartime ago or less before firing is increased by a constant not depending on exact moment of spike arrival. This constant *dH* depends on the basic neuron plasticity value set by the weightinc XML node and current value of a component of neuron state *s* called *stability*. This dependence is expressed by the formula

.

The stability determines the general level of the neuron plasticity. It is also used to fight catastrophic forgetting – but in case of supervised or reinforcement learning. In this tutorial, its values remain low so that it does not significantly influence the learning process. The laws of its dynamics will be described later.

Thus, the synapses helping the neuron to fire are potentiated. This feature helps to find combinations of input nodes, which are often active together, constituting some pattern, for example, the light spot in the center of an image. However, if all neurons recognized the same pattern it would not satisfy us – the different neurons should recognize different patterns. To reach this goal, a special mechanism called “winner-takes-all” (WTA) is commonly used. This mechanism is implemented by interneuron (*lateral*) connections using which neurons inhibit each other. In our case, there are two kinds of these lateral links. The first one is non-plastic inhibitory connections with great negative weight. The respective projection has all-to-all connection policy but it should be noted that reflexive connections are prohibited in our emulator. Using this connection, a neuron with earliest reaction to a new stimulus blocks reaction to it by all other neurons. However, it is not sufficient. The neuron-winner should take care that it will be a winner again and again – when the same pattern will be presented. To guarantee that, the synapses of other neurons which receive spikes belonging to this pattern should be depressed. This is the purpose of the second kind of lateral links – so called *reward* links (type="reward"), although in our case they are rather punishment links. They implement another plasticity mechanism, usually called *reward or dopamine plasticity*. It is also very simple. When a neuron receives a spike via its reward synapse, the resources of all its plastic synapses having obtained a spike not later than time *tD* ago are changed by the value equal to the weight of this reward synapse. This weight may be positive or negative (as it is in our case). *tD* is called *dopamine plasticity period* and is specified by the dopamine\_plasticity\_time XML node.

As it was said, in the present example the neuron stability does not play a significant role. Nevertheless, for sake of completeness, let us describe its dynamics. In this example, stability changes in two cases:

* When neuron fires, the stability increases by the value of weightinc multiplied by the value of stability\_resource\_change\_ratio.
* When the neuron is punished, its stability is changed (decreased) by the value of reward synapse weight multiplied by the value of stability\_resource\_change\_ratio.

We launch the emulation by the command line

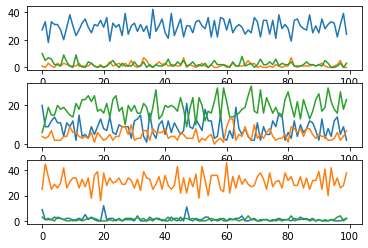
ArNICPU ..\Tutorials -e3 -Pt -v1 –F25000

The additional options are needed to create network monitoring file (-v1). It is needed because we want to explore the plastic synapse weights. The option –F25000 tells that it will be stored every 25000 msec. We would like to explore the network’s final state. Since the total emulation time is 30000 msec, the monitoring at step #25000 is what we need.

During emulation, the system outputs some warning but we can ignore them.

The monitoring data are collected in the file monitoring.3.csv.

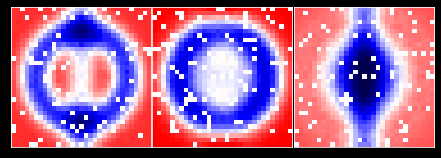
Now let us evaluate the results of unsupervised learning. We break the whole emulation period to 300 msec intervals. In each interval, we count firings of all neurons when each of 3 patterns are presented. In case of successful learning, every neuron should demonstrate systematically higher firing frequency of one pattern and these patterns should be different for different neurons. The respective picture is drawn by the python script ClusteringResults.py in Tutorials. The resulting plot is the following:



Here, every individual plot corresponds to one neuron, one color corresponds to a pattern.

We see that satisfactory pattern discrimination is reached almost instantly and is supported during the whole emulation period. Of course, this task is not very difficult; however, the network is also extremely simple.

At last, let us explore results of learning in the form of the final values of plastic weights. Their values are contained in the monitoring file monitoring.3.csv. It is a text comma-separated-values (CSV) file with complex structure partially described in the manual. The python script file ClusteringWeights.py parses it, extracts values of resources of plastic synapses and depicts them in the form of thermal diagrams corresponding to pixels of input images:



We see that distribution of synaptic weights of recognizing neurons is quite reasonable and corresponds to the classes recognized. The leftmost neuron recognized the central spot. We see that central pixels have great weights except the pixels on the central vertical stipe. The middle neuron recognizes vertical stripe. It has the less evident distribution of weights because its pattern has strong intersection with two other patterns. We see negative weights on the ring (the 3rd pattern) and vertically extended area of moderate positive weights in the center. This neuron acts in accordance with the principle “if nobody then I” – it fires when the other neurons are inhibited by spikes from its pattern. The rightmost neurons recognized the ring and its weighs are a “negative” of two other patterns.

# Neuron Model

The neuron model implemented in this package is a generalized version of the simple but functionally rich model called LIFAT (Leaky Integrate-and-Fire neuron with Adaptive Threshold). Implementation of other models (e.g. Izhikevich neuron) is also possible but requires programming and, therefore, is outside of scope of this manual. LIFAT model itself is less functional than Izhikevich’s model, which can describe several qualitatively different neuron operation regimes, however, additional features introduced by us into it diminish this difference while retaining our model significantly simpler than Izhikevich’s. Besides that, the LIFAT model is implemented in the most advanced modern neuroprocessor Loihi (by Intel Corp.).

Let us describe this model formally, but, at first, consider the synapse model. The simplest current-based delta synapse model is used for all excitatory and inhibitory synapses. Every time the synapse receives a spike, it instantly changes the membrane potential by the value of its synaptic weight, which may be positive or negative depending on the synapse type. The neuron state at any moment *t* is described by its membrane potential *u(t)* and its threshold potential *uTHR(t)*. Dynamics of these values are defined by the equations

and the conditions that *u* is hard limited from below by the value *uMIN* and that if *u* exceeds *uTHR* then the neuron fires and value of *u* is reset to 0. All potentials are rescaled so that after the long absence of presynaptic spikes *u* → 0 and *uTHR* → *H* = 8.531 (see the Tutorials for the discussion of this value). The meaning of the other symbols in the formula above is the following: τ*v* – the membrane leakage time constant; *a* – the speed of decreasing *uTHR* to its base value *H*; *wi* - the weight of *i*-th synapse; *tij* - the time moment when *i*-th synapse received *j*-th spike; – *uTHR* is incremented by this value when the neuron fires at the moment . It should be noted that this model should be rather called linearized LIFAT because threshold potential falls linearly, not exponentially. This feature makes hardware implementation simpler without noticeable impact on network behavior.

Out implementation of LIFAT has two additional feature. Firstly, the memory property is added to this model. Neuron has the parameter called memory spike train period τ*M*. If this parameter is defined (not equal to infinity), then after every firing, the neuron internal timer is reset to the value τ*M*. When this timer reaches zero value, *u* is increased by a great constant. It is significantly higher than H (30 – in the current ArNI-X version) and, therefore, the neuron is forced to fire unless its current membrane potential is too high. It is equivalent to presence of a very strong reflexive connection with the delay time equal to τ*M* and the weight equal to 30 (it is prohibited in our package to create such reflexive connections explicitly). This feature in combination with threshold potential adaptivity allows implementing the mechanism of short-term memory with controlled duration. Indeed, if , then at the moment of the timer reset, *uTHR* becomes high and high. Eventually, it becomes so high that even this imaginary strong reflexive connection cannot make the neuron fire. Therefore, the neuron can memorize that it received strong stimulation in the past, which forced it to fire, but the memory about it may last only a certain time interval.

The other additional feature is the gating ability. Neurons have a state component called the *activation counter A* controlled by spikes coming at special *gating* synapses. When *A* is positive, the neuron is in its ordinary state and behaves as it is described above. If *A* is zero or negative then the neuron is in *sleepy* state. It means that it does not react to any incoming spikes (except spikes coming to gating synapses) and is not able to fire. While it is non-zero, it is changed by 1 towards 0. If *A* was 1 and becomes 0, it remains equal to 0 for indefinite time. If *A* was -1 and becomes 0 it is reset to a very great positive number (= +∞). Neuron may have synapses, which can change *A* (gating synapses). Their weight may be either negative or positive. If neuron receives a spike via a gating synapse with the negative weight and the current value of *A* is greater than that weight, *A* is set to the value of that weight. Therefore, in this case, the neuron becomes inactive and remains in this state the time equal to the absolute value of weight of that gating synapse (in msec). If the receiving gating synapse has positive weight and the current *A* is less than that weight, *A* is also set to the value of that weight. All this means that gating synapses can either activate neuron for the specified period or, conversely, block its activity, thus performing gating functions. Gating synapses can be considered as an ultimate version of strong excitatory or inhibitory synapses but with exactly controlled temporal characteristics and more deterministic effect on neuron state.

# synaptic plasticity rules

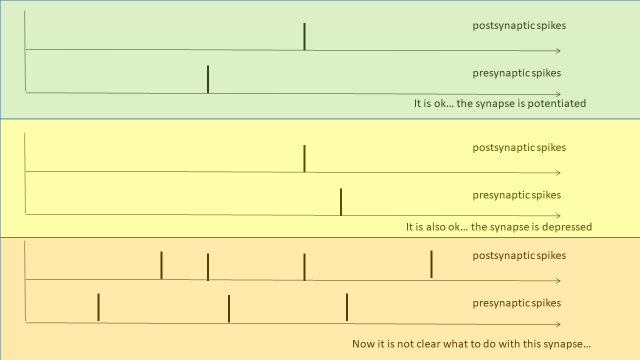
The most valuable ability of neural networks is their ability to learn. Learning in the traditional non-spiking neural networks (artificial neural networks – ANN) is implemented due to the appropriate modification of synaptic weights of their neurons. In this sense, the SNNs do not differ from the ANNs – their synaptic weights are also adjusted during the learning process. However, the approaches to synaptic weight modification in ANN and SNN are completely different. Output value of ANNs is in fact a smooth function of its synaptic weights. It makes it possible to apply a gradient descent technique (the so-called backpropagation algorithm) to optimization of the synaptic weights. In contrast with them, SNNs are discrete by their nature. They produce spikes instead of real numbers. Therefore, the gradient descent algorithm cannot be used for SNN - the respective partial derivatives cannot be calculated. For this reason, the learning of SNN is based on completely different principles. The basic one is the locality principle reflecting discrete and asynchronous functioning of spiking neurons. It stipulates that modification of a synaptic weight must depend on properties and activity of the pre- and post-synaptic neurons.

In ArNI-X, excitatory and inhibitory synapses are plastic only if they are explicitly declared as plastic. Otherwise (by default), they are not plastic (*fixed*) – the plasticity rules do not affect them. Difference between plastic and fixed synapses is important as they play the different roles. Plastic synapses are connected to sources of signal conveying information about the external world used for learning. Fixed synapses are usually strong and used for some special needs when it is necessary to force a neuron to fire. Thus, if a neuron fires when it receives a spike via its fixed synapse, we will call it *forced* firing.

Three kinds of local synaptic plasticity rules are available in ArNI-X without additional programming. All them are very simple and designed with a view to their efficient implementation in neuromorphic hardware. One of them works with the ordinary plastic excitatory and inhibitory synapses while the two others require presence of the special plasticity-modulating (*reward* or *dopamine*) synapses in the neuron. It is necessary to remind that, as it was discussed in the Tutorial, the plasticity rules are applied to value of synaptic resource instead of synaptic weight. The synaptic weight values are updated (calculated from the current synaptic resource value) when the changes of synaptic resources become too great. In addition, it should be noted that usually these plasticity rules act in combination. Let us consider them.

## Hebbian Plasticity

The synaptic plasticity principle formulated by Donald Hebb claims that all synapses that helped the neuron to fire are strengthened. This principle got its empirical confirmation in the form of the STDP (Spike Timing Dependent Plasticity) plasticity model discovered in the end of last century in living neurons. In accordance with this rule, the synapses obtaining spikes short time before firing are potentiated, but if a synapse obtains a spike short time after firing, it is depressed. The rule is simple and useful; however, it becomes self-contradictory in the case of frequent firing as it is shown on the picture below.



It is why we bind the plasticity rule to postsynaptic spike trains instead of single postsynaptic spikes. We will refer to these spike trains as *tight spike sequences* (TSS) – saying about postsynaptic spikes emitted by the given neuron. Specifically, taking the constant *ISImax* (ISI = Inter-Spike Interval) as a measure of “tightness” of TSS, we define TSS as a sequence of spikes adhering to the following criteria:

1. There were no spikes during time *ISImax* before the first spike in TSS;
2. Inter-spike intervals for all neighboring spikes in TSS are not greater than *ISImax*;
3. There are no spikes during time *ISImax* after the last spike in TSS.
4. Forced firing terminates the current TSS. A stand-alone forced firing is also considered as a particular case of TSS however, as we will see, forced firing is treated by the plasticity rules in a special way.

The STDP and the similar rules are often used for unsupervised learning. The goal is to find a group of synapses, which often obtain spikes inside the same sufficiently narrow time window. If the spikes come almost simultaneously, the resulting membrane potential increase is sufficient for firing and as the result of Hebbian plasticity the participating synapses get more strength and their common participation in next firing becomes more probable. The standard STDP rule seems to fit this purpose very well, but after closer investigation, one its drawback becomes evident. In the beginning of the learning process, this rule works well. It really potentiates the synapses from that group of correlating synapses obtaining spikes inside the same time interval. But as their weight grow, the neuron begins to fire earlier inside this interval. Therefore, there will be synapses from the same group that obtain spikes after firing. In accordance with STDP, they will be suppressed (and strongly suppressed) although they should be strengthened. This negative effect leads to unstable learning. To overcome it we introduce the symmetric version of STDP. In our model, any synapse receiving a spike at the moment close to firing (no matter – before or after) is potentiated. Besides that, it eliminates the above mentioned inconsistency of the standard STDP in case of frequent firing.

With the described amendments (association with a TSS instead of a stand-alone postsynaptic spike and symmetry with respect to postsynaptic spikes), our synaptic plasticity model follows Hebb’s principle. Namely, it includes the following rules:

1. The resource of any synapse can change at most once during a single TSS.
2. The resources of only those synapses are increased which receive at least one spike during TSS or short time *TH* before the very first spike in the TSS. Goal of this rule is to reward all the synapses which contributed to postsynaptic spikes in the given TSS. Therefore, the synapses having obtained spikes shortly before the TSS onset should be also rewarded. Effect of one spike to membrane potential decays with the time constant τ*v*, hence *TH* should be few times greater than τ*v* – we selected *TH* = 3τ*v*.
3. All synaptic resources are changed by the same value *dH* independently of exact timing of presynaptic spikes.

It should be added that *dH* may be negative as well as positive – thus, anti-Hebbian plasticity is also possible in our system.

## 2-Factor Dopamine Plasticity

In case of Hebbian (or anti-Hebbian) plasticity, the direction of synaptic resource modification is always the same for the given neuron. However, in many learning tasks, more flexible weight adjustment is needed – when a neuron behaves correctly, the active synapses should be potentiated, otherwise they should be depressed.

This flexible synaptic weight regulation is performed with help of special synapses called reward or dopamine synapses. These synapses may have positive or negative weight and spikes coming to them can increase or decrease synaptic resources of plastic synapses by the value proportional to their weight. So that the name “reward synapses” is not quite correct – these synapses may “reward” as well as “punish” plastic synapses.

There are two kind of plasticity rules using reward synapses in ArNI-X. First, we consider the simpler one called 2-factor dopamine plasticity rule because it is based on two types of events – obtaining a spike by a plastic synapse and obtaining a spike by a reward synapse.

2-factor dopamine plasticity rule is very simple. Whenever a reward synapse obtains a spike, the synaptic resources of all plastic synapses having obtained at least one spike during last *TD* msec are changed by a value proportional to the weight of this reward synapse (or to the sum *R* of weights of all reward synapses obtaining a spike). About the proportionality coefficient – see below. This rule has one exception – it is not applied if the most recent neuron firing was forced and *R* ≤ 0.

## 3-Factor Dopamine Plasticity

3-factor dopamine plasticity rule includes one more event that should happen in order to trigger weight changes – the neuron firing. More precisely, similar to Hebbian plasticity, 3-factor dopamine plasticity rule is associated with a TSS instead of a single postsynaptic spike (which is a particular case of a TSS). 3-factor dopamine plasticity is triggered by a spike incoming to a reward synapse but only if the neuron fired not more than *TD* msec ago. If it was a forced firing, this plasticity rule works only in the case of the positive total reward *R*. 3-factor dopamine plasticity rule changes the resources of all synapses contributing to the last TSS by the value proportional to the weight of the reward synapse. The meaning of the word “contributing” is the same as for Hebbian plasticity. These are the synapses which received at least one spike during the last TSS or the time *TH* = 3τ*v* before the very first spike in the TSS.

Hebbian plasticity and one type of dopamine plasticity can be combined inside one neuron. However, one neuron can have only one type of dopamine plasticity.

## Synaptic Resource Renormalization

In order to strengthen competition between synapses, we introduced the postulate that the total synaptic resource of one neuron should not change in time. It means that if the resource of some synapse is increased then the resources of all other synapses should be appropriately decreased by an equal value. However, effect of this mechanism can be regulated by the following mechanism. A neuron may have some number of unconnected plastic synapses. They have no presynaptic neurons and serve only as a reservoir for excessive synaptic resource (or as a source of synaptic resource deployed on the working plastic synapses). The renormalization procedure is invoked when amount of the synaptic resource to re-distribute exceeds a certain threshold.

## Neuron Stability

In the Tutorial devoted to unsupervised learning, we mentioned the problem of catastrophic forgetting and the concept of synaptic resource as a method to fight it. This method is efficient in the case of unsupervised learning because the asymptotic state of synapses of trained neurons in this case is saturated – close either to minimum (often – zero) or to maximum. However, supervised learning often requires exact setting of synaptic weights far from their minimum and maximum values. Therefore, an alternative mechanism for preventing catastrophic forgetting is needed. In order to implement such a mechanism, an additional component of neuron state is introduced. It is called the *stability s*. This value determines synaptic resource changes caused by the synaptic plasticity mechanisms considered above. The neuron plasticity falls exponentially with growth of *s*. For untrained neurons, *s* = 0.

For example, the resource change *dH* in the Hebbian plasticity rule depends on *s* this way:

Here, is the basic level of Hebbian plasticity.

For dopamine plasticity, the resource change is , where *D* is the weight of the reward synapse obtaining the spike.

The rules controlling changes of *s* are following:

1. **Non-positive *s* can only increase.**
2. The first firing in TSS (and the forced firing) changes *s* by , where *r* is a constant.
3. When a neuron receives reward spikes, its stability changes accordingly to the table:

|  |  |  |  |
| --- | --- | --- | --- |
| The dopamine plasticity rule | The most recent firing was forced | The *R* sign | The stability change |
| 2 factor | no | negative | *rR* |
| 2 factor | no | positive | , where *tTSS* – the time since the first firing in the most recent TSS |
| 2 factor | yes | negative | 0 |
| 2 factor | yes | positive | -*rR* |
| 3 factor | no | negative | *rR* |
| 3 factor | no | positive | 2*rR* |
| 3 factor | yes | negative | 0 |
| 3 factor | yes | positive | -*rR* |

These rules may seem too complicated but they have a natural explanation. For example, since in our system, synaptic weights change discretely, the number of stability change acts should be proportional to the number of weight changes – it is why all stability change formulae have the same factor *r*. The form of these formulae follows from the learning purpose. For example, if some neuron should learn to fire at the correct moment then the four situations are possible:

* **The neuron did not fire and it was right.** Nothing happens.
* **The neuron did not fire but was expected to fire.** In this case, we force the neuron to fire using the strong fixed excitatory synapse. All plastic synapses, which would help it to fire (the synapses having obtained spikes recently), are potentiated due to 3-factor plasticity. But this neuron behavior shows that it is not trained yet. Therefore, to facilitate its further training, its stability should be decreased.
* **The neuron fired but it was wrong.** When the neuron fired it was not clear was it right or wrong. It is safer to think that it was wrong – because if this neuron will not receive reward in the near future, then it was a wrong firing. Therefore, when the neuron fires, the contributing synapses should be depressed and its stability lowered.
* **The neuron fired and it was right.** In this case, it receives reward short time after it fired. Its weights should not change (because it has learnt already – it performs well) and its stability should be elevated. The later requirement is satisfied because in this case the stability increment is doubled – 2*rR*.

It can be shown that the rules described above are consistent with the purposes of unsupervised and supervised learning.

In the next chapter, we will show how the parameters of neuron model and synaptic plasticity model can be specified on the level of neuron populations and projections using NNC files.

# Network Structure Description

The emulated SNN structure is defined in a special Neural Network Configuration (NNC) file whose name should have the format <EmulationNo>.nnc, where EmulationNo – is an integer number. This file is a text file written in XML language. This chapter describes its syntax.

## Overall Structure of NNC Files

The first line of an NNC file should be

<?xml version="1.0" encoding="utf - 8"?>

It says that it is an XML file with UTF-8 character encoding.

The file should contain one highest level XML node with the tag SNN.

Inside the SNN node, there should be one or more RECEPTORS nodes and one NETWORK node. The former describe input nodes (there may be several input node sections sending signals with different semantics), the latter – the network itself. There may be other kinds of nodes inside SNN (for example, describing what to do with spikes emitted by neurons in the network) but their usage requires accessing API and is not covered by this manual.

## RECEPTORS – Description of Input Nodes

А RECEPTORS node should have two attributes – name and n. The former defines name of the input node section, the latter specifies number of input nodes in this section.

А RECEPTORS node should include only one node with the tag Implementation. It should have only one attribute lib that specifies the dynamic library responsible for generation of input spikes. In this manual we consider only one such library – fromFile, which can read input spikes from a file and add to them Poissonian noise.

An Implementation node should include only one node with the tag args. This node contains various parameters of the input node section (all they are optional) in form of sub-nodes. They are:

* history\_length – number of iterations *T* (= msec) during which the input node section generates the input signal. This parameters is obligatory if the input node section emits pure noise – without reading spikes from a file. If *T* is less than the number of iterations for which input spikes are contained in the file then only *T* first input signal portions are read from the file. If *T* is greater than the number of iterations for which input spikes are contained in the file and signal generation regime (see below) is normal, then *T* is made equal to number of input signal portions contained in the file. **When input signal cannot be produced anymore, the emulation terminates.**
* meanings **–** name of the text file containing labels of the input nodes. This file should contain number of lines equal to number of input nodes – each line should contain identifier of the respective input node which will be displayed in the network activity log file (if the command line argument –v2 is used). By default, the input node label is the input node section named followed by the ordinal number of the node inside the section.
* mode – this parameter can take the values repetitive or endless. If this XML node is present and its value is repetitive, then the signal recorded in the file (if its duration is less than the value of history\_length) is read from the beginning again after it is completely read – until the total length of the input signal reproduced is equal to history\_length. The endless mode means that after the input signal from the file is completely read, the input signal generation can continue (if history\_length is greater than the recorded signal length) but it does not contain spikes (silent input signal).
* noise – Poisson noise intensity *f* (kHz). If this parameter is present, the Poisson noise is added to the input signal. It means that for each input node and each iteration, the random number uniformly distributed in [0, 1] is generated. If it is less than *f*, a spike is emitted.
* period – spike emission period *P* (msec). If this parameter is present, the first input node emits a spike every *P*th iteration – in addition to spikes from the file and noisy spikes.
* source – input signal file name. The file format may be either text or binary – it is inferred from the file extension (binary files should have the .bin extension). In text file, one line corresponds to one iteration. The length of every line should be equal to the input node count. If the given input node should emit spike then that corresponding position in the file should be occupied by ‘@’ character. Otherwise, it should be ‘.’ character. Binary file contains input signal in the form of bit mask – one bit for one input node – one after one. Bit record for one iteration (one input signal portion) should have 64 bit (8 byte) alignment. Bit records for successive iteration go tightly one after one. There also exists a special kind of binary input signal file – image files. They are considered below.
* Special – if present, this node specifies a special format of the input signal file (determined by the type attribute). At present, only one type is supported - image. In this case, the input signal file contains monochrome images. Every image is presented during a certain time period. Every pixel corresponds to one input node. Number of spikes emitted by this input node during this period is proportional to the pixel brightness *b*. The input nodes produce spikes by the following algorithm. Every input node has a state variable associated with it. Its initial value is 0. Every emulation step, it is incremented by the value *gb*, where *g* is constant. If its value reaches 1, then input node fires the the state variable is decremented by 1. The Special node should include the following sub-nodes:
  + height – the image height (pixels).
  + maxfrequency (optional) – the maximum input node firing frequency (kHz). Since brightest pixels have value 255, the constant *g* equals to this values divided by 255. The default value is 1.
  + ntact\_per\_image - time of presentation of one image (msec).
  + offset (optional) – position of the first image in the file (bytes). Default = 0.
  + width - the image width (pixels).
* tacts\_to\_skip – the number of input signal portions (iteration steps) recorded in the file which should be skipped (from the file beginning).

## NETWORK – Description of SNN Structure

In this manual, it is assumed that the NETWORK node includes only one node with the tag Sections. In fact, the NETWORK node can include other kinds of sub-nodes but they correspond to SNN sections implemented by dynamic libraries using the respective API, which is not covered by this manual, and, therefore, are not considered here.

The Sections node includes at least one Section node and at least one Link node. The Section nodes describe neuronal populations, the Link nodes – projections (set of connections between populations).

### Section – Neuronal Population Property Definition

Any population should have a unique name specified by the attribute name.

The population’s neuron properties are specified in the sub-node props.

Моделирование ИНС проходит в дискретном времени. Как в большинстве работ по моделированию ИНС, один квант времени (еще называемый *тактом*) принимается равным миллисекунде (мсек).

Пакет содержит следующие логические блоки:

* **Собственно, модуль эмуляции ИНС.** Все модели нейронов имеют 2 реализации – на GPU и CPU. Это сделано, в том числе, в целях верификации корректности моделирования. Предполагается, что каждая модель синаптической пластичности должна быть реализована и на GPU, и на CPU и проходить этап верификации с помощью сравнения общих параметров функционирования реализации сети на GPU и на CPU. ВАЖНО! Абсолютное совпадение протоколов активности сети на GPU и на CPU невозможно и не предполагается. В процессе моделирования осуществляется интерпретация конфигурационного файла сети, построение сети в соответствии с ним, запуск моделирования, мониторинг активности сети с сохранением его результатов на диске. Так как выбор модели синапсов производится с помощью условной компиляции, значительная часть логики моделирования реализована в заголовочном файле **LIF.inc.h**, а не в библиотеках. Процедуры общего характера, нужные для создания и работы модели сети, которые не зависят от модели нейрона, собраны в статической библиотеке **ArNI**.
* **Генератор входных сигналов.** Генераторами входных сигналов служат динамические библиотеки со стандартным протоколом функций, загружаемые в процессе выполнения. Какие источники эмулируемых данных и с какими параметрами будут использоваться в данном запуске эмуляции ИНС, определяется, как и другие характеристики сети, в специальном конфигурационном файле сети. В поставку пакета входит источник сигналов, читающий их из файла.
* **Интерпретатор активности сети.** Спайки, генерируемые определенной популяцией нейронов, могут рассматриваться в качестве выходных сигналов сети в каком-либо смысле. Они передаются в загружаемую динамическую библиотеку (также заданную в конфигурационном файле), которая как-то их интерпретирует, использует как команды или как-то еще. В конце моделирование эта библиотека также выдает некоторую числовую оценку того, насколько моделирование было успешным в терминах некоторой поставленной задачи.
* **Модули обработки и интерпретации протоколов активности сети.** Имеется в виду статистическая обработка записей активности нейронов, анализ конечного состояния сети с целью понимания происходящих в сети процессов, оценки эффективности решения сетью тех или иных задач, а также графическая визуализация этих результатов.
* **Модули оптимизации параметров сети.** В силу трудности построения теоретических моделей, связывающих параметры отдельных нейронов и синапсов с параметрами сети в целом, оптимальные значения первых приходится искать с помощью некоторой оптимизационной процедуры. Для этого используется генетический алгоритм и координатный спуск.
* Вспомогательные процедуры для отладки и тестирования.

## Системные требования

* **Аппаратура**: мощный PC с 64битным процессором, аналогичным по мощности Intel Core i7, 64GB RAM, 1TB дискового пространства, NVIDIA GPU с compute capability не ниже 5.
* **Операционная система**: MS Windows 10 или выше, Ubuntu 18.04 или выше.
* **Компиляторы (минимальные версии)**: MS Visual Studio 2017, GNU v 6, CUDA Toolkit v 10.2, CMake v 3.18.2, Anaconda v 3.
* **Сторонние библиотеки**: boost v 1.66, pugixml 1.10

## Установка пакета

1. Установить требующиеся компиляторы и библиотеки. Для облегчения этого процесса под Linux предоставляется скрипт ArNI.preinstall.sh. Тем не менее, для корректного выполнения этого скрипта в некоторых случаях требуется его ручная правка. В скрипте надо установить правильный CUDA\_COMPUTE\_CAPABILITY – он должен быть равен минимальному значению для установленных на компьютере GPU.
2. Развернуть из архива ArNI.rar корневую директорию ArNI.
3. Скопировать в нее пакет pugixml (директорию pugixml-1.10).
4. В Linux - cконфигурировать и запустить сборку из директории ArNI/build.

# Модели нейронов и синапсов

В используемой модели нейрон может находиться в двух состояниях – активном и пассивном. Это определяется значением счетчика активности, который уменьшается на 1 (если он больше 0) за каждый шаг моделирования. Нейрон активен, пока этот счетчик больше 0. Этот счетчик устанавливается равным весу специального типа синапса (активирующего синапса) при приходе на этот синапс спайка. Если нейрон пассивен, на него не оказывают никакого влияния спайки, приходящие на любые его синапсы кроме активирующих. Кроме счетчика активности состояние нейрона определяется следующими основными переменными (в описании ниже будут фигурировать также символические идентификаторы параметров модели нейрона с префиксом p\_, применяемые для доступа к этим параметрам из API):

*u* – мембранный потенциал:

*θ* – вариативная компонента порогового потенциала;

*b* – фаза непрерывной генерации спайков.

Изменяются эти величины по следующим законам (за одну итерацию моделирования)

, где

для активного нейрона и 0 – для пассивного, *w+* и *w-* - это веса возбуждающих и тормозных синапсов, соответственно; *s* равные 1 или 0 индицируют приход или отсутствие пресинаптического спайка в данный момент на данном синапсе; α – константа распада мембранного потенциала (p\_BasePotentialDecayRate), *wI* – вес рефлексивной связи.

Если в данную итерацию *u* > *Θ* + *θ*, где *Θ* – базовое значение порогового потенциала, то нейрон генерирует спайк, а *u* устанавливается равным 0.

Пороговый потенциал меняется следующим образом:

, где *Dθ* – константа (p\_ThresholdExcessDecrement). Кроме того, если нейрон в данную итерацию испустил спайк, *θ* увеличивается на константу *ΘIθ* (p\_ThresholdExcessIncrement).

Если переменная b > 0, то она уменьшается на 1 за итерацию. При этом, если нейрон в данную итерацию генерирует спайк, b устанавливается равным *TI* (p\_BurstingPeriod) – периоду генерации спайков в последовательности спайков. Это эквивалентно наличию возбуждающей связи от нейрона к самому себе с весом *wI* и задержкой *TI*.

В данной системе предусмотрено 2 основных модели синаптической пластичности, выбор которых определяется значением макросов и условной компиляцией, - классическая модель STDP и так называемая симметричная STDP. Оба варианта пластичности аддитивно применяются не к синаптическому весу *w*, а к так называемому синаптическому ресурсу *W*, связанному с весом по формуле

.

В то время как *W* меняется в пределах (-∞, ∞), *w* меняется в пределах [*wmin, wmax*).

В классической модели STDP временные константы для увеличения и уменьшения синаптического ресурса одинаковы и равны *TP* (p\_PlasticityPeriod). Величины максимального увеличения и уменьшения синаптического ресурса равны *βA*+ и *A*-(p\_SynapticPlasticityLTD), соответственно, где *β* – переменная, которая может быть применена для подавления усиления весов у нейронов, часто генерирующих спайки. А именно, после каждой генерации спайка нейроном величина *β* уменьшается на *Dβ*, а при отсутствии спайков на каждом шаге эмуляции *β* увеличивается на *Iβ* пока не достигнет 1.

В симметричном STDP действуют следующие правила изменения весов.

1. При приходе спайка синаптический ресурс уменьшается на величину *A*-, но это происходит не более одного раза в любом интервале времени длиной 2*TP*.
2. Первый спайк, приходящий на данный синапс во время *периода пластичности*, вызывает увеличение его синаптического ресурса на *A*+(p\_SynapticPlasticity). Периодом пластичности называется временной интервал длиной 2*TP* с центром в моменте генерации нейроном спайка, но только если этот момент отстоит от центра предыдущего периода пластичности не менее, чем время *TP*. Надо обратить внимание на, что в отличие от классического STDP, синапс усиливается и в том случае, если пресинаптический спайк приходит сразу после генерации нейроном спайка.
3. Если синапс с положительным ресурсом не получает спайка в течение периода пластичности, его ресурс уменьшается на *a*-.

Кроме того, опционально эти модели пластичности могут быть дополнены следующими правилами:

1. Суммарный синаптический ресурс нейрона должен оставаться постоянным. Это значит, что, если в соответствии с описанными выше правилами какой-то ресурс какого-то синапса увеличивается или уменьшается, ресурсы остальных пластичных синапс на равную величину уменьшаются или увеличиваются, соответственно.
2. Нейрон может иметь так называемые *синапсы, индуцирующие пластичность* (далее называемые еще reward синапсы). При приходе спайка на такой синапс ресурсы все пластичных синапсов, получивших спайки в течение времени *TP* до этого, изменяются на величину веса данного синапса, индуцирующего пластичность.

Пластичными могут быть только возбуждающие связи.

Кроме синаптической пластичности изменчивость ИНС выражается также в том, что нейроны могут умирать и рождаться или мигрировать. Это происходит в том случае, если нейрон не генерирует спайков слишком долго или если он слишком долго генерирует спайки слишком интенсивно. Критерии выполнениях обоих условий для перерождения нейрона определяются параметрами его модели. Например, для реализации гибели нейрона от слишком интенсивной генерации спайков используется переменная в его состоянии, которая при каждой генерации спайка увеличивается на p\_TirednessIncrement, но каждую итерации снижается на 1. При достижении этой переменной значения p\_CriticalTiredness, нейрон может мигрировать. Также, нейрон может мигрировать, если не генерирует спайков в течение p\_SilentDeathPeriod шагов итерации.

# Описание структуры сети

Структура моделируемой ИНС (источник входных сигналов, секции сети и их свойства, связи между секциями и т.д.) задается комбинацией двух способов. Она в целом или ее компоненты описываются в специальном текстовом файле в формате XML с именем <число>.nnc. Кроме того, структура ИНС может дополняться и модифицироваться вызовами API, предоставляемого модулем эмуляции специальной загружаемой динамической библиотеке, задаваемой в конфигурационном файле. Заметим, что используемая модель нейрона и синапсов НЕ задается этими способами (могут быть заданы только ее параметры – в том числе, на уровне отдельного нейрона)– она задается при сборке программы-эмулятора средствами условной компиляции.

Содержимое всего файла входит в раздел Network.

В нем есть разделы Receptors (их может быть несколько), Sections и Readout. Первые описывает источники спайковых сигналов, второй – саму сеть, третий – модуль, как-то использующий или интерпретирующий активность определенных нейронов сети. Раздела Readout может не быть. Кроме того, могут наличествовать секции Global, содержащие число. Эти числа задают некоторые глобальные параметры алгоритма моделирования (они вставляются в вектор vd\_Globals). В текущей версии используется лишь один такой параметр, задающий отношение *a*- / *A*+.

У раздела Receptors есть атрибуты name – имя источника сигналов и N – число источников. В нем есть раздел Implementation, описывающие реализацию этого источника. Атрибутом этого раздела является имя реализующей его динамической библиотеки. Программный интерфейс таких библиотек и стандартная динамическая библиотека fromFile, реализующая чтение входных сигналов из файла, описаны в следующем разделе. В разделе Receptors может находиться раздел args, содержащий параметры генератора спайкового сигнала, специфичные для конкретного генератора.

Раздел Sections может иметь 2 формата. В первом формате предполагается, что ИНС конфигурируется загружаемой динамической библиотекой. Этот формат совпадает с форматом раздела Receptors, но без атрибута N. Второй формат включает один или несколько разделов Section, описывающих популяции нейронов в сети с одинаковыми свойствами, и может включать разделы Link, описывающие связи (проекции) между популяциями.

Раздел Section включает атрибут – имя популяции и раздел props, описывающий свойства ее нейронов. Перед именем популяции стоит знак +, - или \*, определяющий, являются ли постсинаптические связи данной популяции по умолчанию возбуждающими, тормозными или индуцирующими пластичность.

Параметры, описанные в разделе props, определяют свойства нейронов в этой популяции и один обязательный параметр N, задающий число нейронов в популяции. Кроме того, в разделе Structure может быть определена пространственная структура популяции. Например, если атрибут Type этого раздела равен S, то нейроны занимают случайные позиции на N-мерной сфере единичного радиуса, где размерность сферы задается атрибутом Dimension. Если этот атрибут равен W, то эта популяция имеет структуру WTA (winner-takes-all – победитель получает все). В популяции такого типа все нейроны связаны тормозными связями с одинаковым весом p\_WTAWeight, при этом испускание спайка одним нейроном исключает испускание спайков любым другим нейроном этой же популяции в данном шаге эмуляции. Значение Type, равное «O», задает «колончатую» структуру популяции. В этом случае атрибут Dimension задает количество «колонок». Колончатая структура определяет структуру нейронных связей – внутри одной популяции возбуждающие связи могут только между нейронами одной колонки, а тормозные – только между нейронами разных колонок; между разными популяциями со структурой O и одним и тем же количеством колонок любые связи могут быть только между нейронами с одним и тем же номером колонки в популяции.

Другие параметры этой секции следующим образом соотносятся с константами модели нейрона, описанными в предыдущем разделе:

CHARTIME – временная константа релаксации мембранного потенциала τ – α = 1 – 1 / τ.

WEIGHTINC - *A*+

RELWEIGHTDEC – *A*- / *A*+

THRESHOLDINC - *Iθ*

THRESHOLDDECAYPERIOD - *ΘIθ* / *Dθ*

PLASTICITYTIME - *TP*

В этой секции может находиться также секция MIGRATION с возможными значениями –INSIDE, SAME\_PLACE и NONE. В первом случае при перерождении нейрона он должен родиться в той же популяции, во втором случае – c теми же пресинаптическими нейронами, но другим набором постсинаптических нейронов, в третьем случае нейрон не перерождается ни при каких обстоятельставх.

Раздел Link должен включать атрибуты from и to, определяющие пре- и пост-синаптические секции для данной проекции. Кроме того, могут быть определены атрибут type, определяющий тип синапса (excitatory, inhibitory, activating, reward) и атрибут policy, определяющий специальную стратегию связывания нейронов. Его допустимые значения:

one-to-one – один нейрон связывается с одним – по порядку их идентификационных номеров.

all-to-all – все нейроны связываются со всеми.

WTA – связи как в случае all-to-all, но без связей между нейронами с одним и тем же номером нейрона внутри популяции.

Раздел Link включает набор секций, задающих параметры связей, которые зависят от используемой модели синапса. А именно:

DELAY – распределение синаптических задержек, которая может иметь один из двух форматов:

<DELAY Type="UNI">

<min>*dmin*</min>

<max>*dmax*</max>

</DELAY>

, что задает равномерное распределение задержек в интервале [*dmin, dmax*], либо

<DELAY Type="LN">

<mean>*dmax*</mean>

<stddev>*σd*</stddev>

</DELAY>

, что задает логнормальное распределение задержек, так что натуральный логарифм задержек нормально распределен с центром в *dmax* и стандартным отклонением, равным *σd*.

MAXWEIGHT –эта секция задает параметр *wmax* и применяется только для пластичных связей.

WEIGHT – применяется только для непластичных связей, задает вес связи. Для непластичных связей это единственная секция, задающая синаптический вес.

INIWEIGHT – задает распределение значений синаптических ресурсов на начало моделирования (только для пластичных проекций). Эта секция должна иметь формат

<INIWEIGHT Type="DIS">

<default>*D*</default>

<value v="*vi*" share="*si*"></value>

</INIWEIGHT>

Здесь *D* – это умолчательное значение, разделы value (их может быть 0 или несколько) определяют, какой доле *si* связей приписывается синаптический ресурс *vi*.

probability – максимальная вероятность связи между двумя конкретными нейронами из связываемых популяций (если не указана policy). Связь нейрона с самим собой запрещена. Выбор связываемых нейронов производится случайно. В случае описанной пространственной организации связываемых популяций типа S вероятность связей определяется расстоянием между нейронами *d* по формуле *p* = *A*exp(*-(d – d0)2/s2*). *A* – это максимальная вероятность, *d0* и *s* определены в разделе SpatialDistribution данной связи.

У раздела Readout должен быть атрибут lib, задающий имя динамической библиотеки, которая осуществляет обработку спайков нейронов выходной популяции, и подраздел output, содержащий имя этой популяции.

Для синтаксического разбора конфигурационных файлов используется пакет pugixml.

# Эмуляция входного сигнала ИНС

В пакете ArNI-X источником эмулированного входного сигнала являются динамические библиотеки, поддерживающие описанный ниже API.

## Интерфейс генераторов входного сигнала

1. **Задание параметров сигнала.** Эти параметры (как и название самой библиотеки, реализующей эмуляцию входного сигнала) берутся из конфигурационного файла сети, описанного в предыдущем разделе.

IReceptors \*SetParameters(int &nReceptors, const pugi::xml\_node &xn)

nReceptors– число входных узлов, оно задается извне, но может быть скорректировано в процессе инициализации библиотеки,

xn – представления xml-узла, задающего параметры источника сигналов. Их интерпретация специфична для конкретного генератора входного сигнала.

Возвращает эта функция указатель на более детальный интерфейс генератора входного сигнала.

Данная функция вызывается один раз первой после загрузки динамической библиотеки, в нее может быть включен какой-либо инициализационный код.

1. **Загрузка состояния генератора сигнала.** Как часть процедуры загрузки ранее сохраненной сети состояние генератора сигнала также может быть загружено из сериализатора с помощью вызова функции

IReceptors \*LoadStatus(Serializer &ser)

Для класса сериализатора реализованы операторы сериализации >> и << для основных типов и контейнеров. Сохранение состояния генератора входных сигналов реализуется виртуальным методом IReceptors::SaveStatus(Serializer &ser).

Наследник IReceptors должен реализовать два его виртуальных метода (кроме SaveStatus):

* 1. **Рандомизация процесса генерации входного сигнала.** Это бывает нужно для проведения серии экспериментов с одинаковыми по общим свойствам входными сигналами, но не идентичными.

void Randomize(void)

Данная функция вызывается один раз после SetParameters, если при запуске программы эмуляции ИНС была выбрана рандомизация.

* 1. Генерация входных сигналов.

bool GenerateSignalsforTact(char \*prec, size\_t neuronstrsize)

Булевский сигнал (спайк) от i-ого рецептора (входного узла) записывается по адресу prec + i \* neuronstrsize

Нормально функция возвращает true. Возврат false означает, что входной сигнал больше не может генерироваться и вычислительный эксперимент должен быть окончен.

Функция вызывается перед каждым очередным квантом времени моделирования.

Кроме того, нужно реализовать деструктор.

# Конфигурирование ИНС

Как было сказано ранее, конфигурирование ИНС может производиться загружаемой динамической библиотекой, заданной в подразделе Implementation раздела Sections и поддерживающей описанный ниже API.

## Интерфейс динамической библиотеки - конфигуратора ИНС

1. **Конфигурирование.** Все необходимое для конфигурирования сети должно содержаться в подразделе args раздела Implementation. Там могут находиться описания популяций проекций и целых фрагментов (см. Описание структуры сети).

void SetParameters(const pugi::xml\_node &xn, const INetworkConfigurator &inc)

xn – представление xml-узла args, задающего структуру сети.

inc – интерфейс конфигуратора сети, описанный ниже.

Данная функция вызывается один раз при инициализации эмуляции.

1. **Задание функций, определяющих смысловое описание нейронов.** Такая функция pfnsetmeanings определяется для каждой популяции, задаваемой ее именем.

void SetMeaningDefinitions(vector<pair<const char \*, pfnsetmeanings> > &vppchfsm\_)

Прототип pfnsetmeanings:

void (\*pfnsetmeanings)(const std::vector<std::vector<std::pair<int, int> > > &vvp\_Synapses, std::vector<std::string> &vstr\_Meanings)

vvp\_Synapses – описание пластичных синапсов нейронов популяции в порядке их идентификаторов. Каждый синапс описывается парой <идентификатор пресинаптического нейрона (эти идентификаторы отрицательны для входных узлов), масштабированный (целочисленный) синаптический ресурс>

vstr\_Meanings – смысловое выражение соответствующего нейрона.

Данная функция вызывается один раз при инициализации эмуляции, а функции pfnsetmeanings – при каждом мониторинге сети (см. ниже).

## Интерфейс конфигуратора ИНС INetworkConfigurator

void LoadNetwork(std::string strSavedNetwork) – загрузить сеть из файла strSavedNetwork и установить все нейроны в начальное состояние.

bool bAddNetwork(const pugi::xml\_node &xn) - добавить сеть, описанную в разделе SECTIONS, соответствующем XML узлу xn.

std::vector<std::string> vstr\_GetSectionNames() – получить имена всех популяций нейронов

std::vector<std::string> vstr\_GetInputNames()– получить имена всех секций входных узлов

std::string strCreatePopulation(const pugi::xml\_node &xn) – создать популяцию нейронов, описываемую XML узлом xn и населить ее нейронами. Вернуть имя популяции.

bool bDuplicatePopulation(std::string strExistingPopulation, std::string strNewPopulation, bool bConnect) – создать копию популяции strExistingPopulation и присвоить ей имя strNewPopulation. Если bConnect равен true, создать все пресинаптические связи новой популяции.

size\_t GetNNeurons() – получить число нейронов в сети.

void GetNeuronIds(std::string strPopulation, std::vector<size\_t> &vind\_) – получить идентификаторы всех нейронов в популяции strPopulation.

void FixSection(std::string strPopulation) – сделать все нейроны в популяции непластичными (и не мигрирующими).

void SetNeuronProperty(size\_t idNeuron, LIFproperty lp, double dval) – установить свойство lp нейрона idNeuron равным dval. Код свойства – это константы с префиксом p\_, описанные ранее.

IntersectionLinkProperties \*pilpCreateProjection(const pugi::xml\_node &xn, IntersectionLinkProperties::connection\_type ct) – создать проекцию, описываемую XML узлом xn и имеющую тип ct (connection\_excitatory, connection\_inhibitory, connection\_reward, connection\_activating). Возвращается указатель, который можно использовать далее как идентификатор проекции. Он должен быть впоследствии уничтожен.

void DestroyProjection(IntersectionLinkProperties \*pilp) – уничтожить описание проекции (сама связи, составляющие проекцию при этом не разрушаются).

bool bConnectPopulations(std::string strPresynapticPopulationName, std::string strPostsynapticPopulationName, const IntersectionLinkProperties \*pilp) – соедини нейроны популяции strPresynapticPopulationName с синапсами нейронов популяции strPostsynapticPopulationName с помощью проекции pilp. Верни false, если это невозможно.

bool bConnectNeurons(int Presynaptic, unsigned Postsynaptic, const IntersectionLinkProperties \*pilp, int ExactScaledWeight = 0) – соедини нейроны с идентификаторами Presynaptic и Postsynaptic в соответствии с правилами проекции pilp и, возможно, задав точный вес проекции ExactScaledWeight (целочисленный масштабированный вес).

void Finalize() – этот метод должен быть вызван последним, когда конфигурирование сети полностью закончено.

# Реакция на активность выходной популяции ИНС

Определенные популяции нейронов могут рассматриваться как выходные – формирующие имеющие спайковую форму сигналы, интерпретируемые как вырабатываемые сетью решения или управляющие команды, с помощью которых сеть воздействует на окружающую среду. Реакция на активность нейронов этих популяций реализуется динамической библиотекой, поддерживающей описанный ниже API.

## Интерфейс интерпретатора выходного сигнала

1. Получение идентификаторов активных нейронов.

bool ObtainOutputSpikes(const vector<int> &v\_Firing, int nEquilibriumPeriods)

v\_Firing – идентификаторы нейронов выходной популяции, которые испустили спайк в текущей итерации эмуляции,

nEquilibriumPeriods – в данной версии не используется.

Если эта функция возвращает false, это значит, что эмуляция должна быть завершена.

Данная функция вызывается после каждого шага эмуляции сети.

1. Обработка и вывод результатов эмуляции.

int Finalize(int OriginalTerminationCode)

OriginalTerminationCode – код завершения эмуляции – отрицательное значение означает ошибку.

Функция вызывается непосредственно перед завершением программы - эмулятора.

# Эмуляция ИНС

## Общая логическая структура компонент, отвечающих за эмуляцию ИНС

Макросы, темплейты и инлайновые функции общего характера определены в заголовочном файле ArNIcommon.h.

Макросы, определяющие модель нейрона, задаются в model.h.

Сам класс нейрона с учетом условной компиляции и определенных в model.h макросов определен в заголовочном файле LIF.hpp.

Все процедуры эмуляции, зависящие от выбранной модели нейрона, определены во включаемом файле LIF.inc.h.

GPU-реализация модели нейрона определена во включаемом CUDA файле LIF.kernel.cuh.

Все процедуры эмуляции, не зависящие от выбранной модели нейрона, определены в библиотеке ArNI. В том числе там находится процедура запуска эмуляции, общая для CPU и GPU реализации – RunEmulation в файле RunEmulation.cpp.

CPU реализация эмуляции ИНС производится программой ArNICPU.

GPU реализация эмуляции ИНС производится программой ArNIGPU.

## Командная строка запуска эмуляции ИНС

Аргументы командной строки, используемые для запуска эмуляции ИНС одинаковы для CPU и GPU реализаций. Командная строка имеет синтаксис:

./ArNI(C|G)PU ../Experiments( -?\*)+

Начиная со второго аргумента, командная строка содержит опции, имеющие формат «тире, буква опции, аргумент опции».

Значения опций и формат их аргументов:

С – выбор номеров устройств, на которых происходит эмуляция. Аргумент имеет либо синтаксис <устройство>(,< устройство >)\* - и в этом случае интерпретируется как список номеров устройств, либо N<число устройств> - в этом случае задается лишь число устройств (система должна сама выбрать самые подходящие).

d – нейроны могут мигрировать. При формате этого аргумента dS<число> это число означает минимальное число тактов молчания, после которого нейрон может мигрировать. По умолчанию оно равно 10000. Минимальное число тактов молчания, после которого нейрон может мигрировать, устанавливается для каждого нейрона случайно, но не может быть меньше этой величины (может быть в несколько раз больше).

D - количество миграций нейронов, при достижений которого эмуляция прекращается и считается неудачной.

E – если аргумент не задан, эмуляция должна остановиться, когда изменения синаптических весов становятся малыми. Если он задан, то это число интерпретируется как номер такта эмуляции, после которого производится оценка общего изменения весов. Если изменение весов остается большим, эмуляция останавливается и трактуется как неудачная.

e – аргумент – это номер конфигурационного файла сети. Эта опция обязательно должна присутствовать.

F – период (в тактах) через который проводится мониторинг синаптических весов.

f – номер такта, на котором синаптическая пластичность выключается.

i – период в тактах, с которым делается проверка на необходимость миграции нейронов. Т.е. в этот период может мигрировать максимум 1 нейрон.

M – предельная частота генерации спайков (kHz). Если частота генерации превышает этот порог, эмуляция останавливается и считается неудачной.

n – число создаваемых сетей с одной и той же структурой (но с разными межнейронными связями).

p – имя трубы, в которую надо выводить протокол активности сети.

R – рандомизация эмуляции. Аргумент имеет формат [S]<затравка>. Затравка используется для рандомизации генератора случайных чисел. При наличии буквы S рандомизируется также входной сигнал.

r – будет также сохранен полный протокол активности входных узлов сети.

S – число тактов. Если на протяжении его ни один возбуждающий нейрон не срабатывает, эмуляция обрывается и считается неудачной.

s – аргумент – список тактов через запятую, в которые текущее состояние сети должно быть сохранено.

T – длительность эмуляции в тактах.

t – управление сохранением протокола активности сети. Аргумент имеет вид <начальный такт сохранения протокола>-<конечный такт сохранения протокола >.

V – потактовое сохранение сигнатуры состояния сети. Аргумент – имя файла, куда сохранять сигнатуры.

v – сохранение подробной информации об эмуляции.

@ - сохранение протокола активности сети в виде текстового файла, где «.» означает отсутствие, а «@» - наличие спайка (строка – номер такта, столбец – номер нейрона).

## Результаты эмуляции

При штатном завершении работы эмулятора он возвращает 0, а в stderr пишет “ArNI<общий результат эмуляции>”. Если интерпретатор выходной активности сети не определен, общим результатом эмуляции является количество выполненных шагов эмуляции. Если он определен, то результат возвращается функцией Finalize интерпретатора.

Результатами эмуляции являются:

1. Протокол активности сети (см. опции t, p, r и @). Если не указана опция @, протокол сохраняется как битовая маска. Если не указана опция p, он сохраняется в файл spikes.< номер конфигурационного файла сети>.(bin|txt). Если указана опция r, в файле receptor\_spikes.< номер конфигурационного файла сети>.(bin|txt) сохраняется также протокол активности входных узлов.
2. Результаты мониторинга сети (см. опцию F). Это файл в формате CSV с именем monitoring.<номер конфигурационного файла сети>.csv. Первое строковое поле этого файла определяет тип записи, формат записи зависит от этого типа.
3. Финальное состояние сети, сохраняемое в файле <номер конфигурационного файла сети>.finsave.nns.
4. Промежуточные состояния сети, если выбрана опция s.
5. Если выбрана опция v, ряд других файлов, содержащих, например, координаты нейронов.
6. Если выбрана опция V - файл CRC-сигнатур состояний сети за каждый такт, одна сигнатура на строке. Применяется для отладки и верификации моделей нейронов.

## GPU реализация – проект ArNIGPU

Основная функциональность GPU версий моделей реализована в файле LIF.kernel.cu в виде 2 рассмотренных ниже GPU процедур, вызываемых последовательно каждый такт. Состояние и свойства всех нейронов хранятся в массиве объединений U32. Состояние одного нейрона должно укладываться в 32 (константа NEURONSOMASIZE\_DW) двойных слова. Состояния и свойства синапсов хранятся в нескольких 2-мерных массивах двойных слов, следующих один за другим. В одном массиве хранится один и тот же параметр для всех синапсов всех нейронов. Размер каждого массива - <максимальное кол-во синапсов у нейрона> X <число всех нейронов>. Сначала идут все синапсы одного нейрона, затем синапсы следующего нейрона и т.д.

Интерфейс с платой GPU осуществляется классом GPUInterface.

Первая из 2 GPU процедур реализует пересчет состояния нейрона и генерацию спайка:

\_\_global\_\_ void kerProcessSynapses(

union U32 \*pDnhc,

unsigned \*pDExcitatorySynapses,

unsigned \*pDInhibitorySpikes,

unsigned \*pDRewardSpikes,

const inhibitory\_weight\_cuda \*piwcDInhibitoryWeights,

const int \*pDRewardWeights,

int MaxNExcitatorySynapses,

int MaxNInhibitorySynapseBlocks,

int MaxNRewardSynapseBlocks,

unsigned ntact,

char \*pDDead,

bool \*pDbPostsynapticSpike,

unsigned RandomBlockShift

)

Процедура обрабатывает пришедшие пресинаптические спайки.

pDnhc – указатель на вектор состояний нейронов.

pDExcitatorySynapses – указатель на массив состояний возбуждающих синапсов (ВАЖНО! Имя этого аргумента не должно меняться).

pDInhibitorySpikes – указатель на массив числа спайков, пришедших на секции тормозных синапсов нейронов.

pDRewardSpikes – указатель на массив числа спайков, пришедших на секции индуцирующих пластичность синапсов нейронов.

piwcDInhibitoryWeights – указатель на массив весов секций индуцирующих пластичность синапсов нейронов.

pDRewardWeights – указатель на массив весов секций тормозных синапсов нейронов.

MaxNExcitatorySynapses – максимальное количество возбуждающих синапсов у нейронов (ВАЖНО! Имя этого аргумента не должно меняться).

MaxNInhibitorySynapseBlocks – максимальное количество секций тормозных синапсов у нейронов.

MaxNRewardSynapseBlocks – максимальное количество секций индуцирующих пластичность синапсов у нейронов.

ntact – номер текущего такта.

pDDead – индикатор смерти нейрона (позиция i + 1 соответствует индексу нейрона i). Значения берутся из перечисления NeuLIF::DeathCause. alive означает, что нейрон жив, silent – что нейрон умер из-за малой частоты генерации спайков, exhausted - что нейрон умер из-за слишком большой частоты генерации спайков. Нулевая позиция этого вектора – это булевское значение, индицирующее, что один из нейронов умер.

pDbPostsynapticSpike – указатель на массив генерируемых спайков. Также, этот массив содержит спайки, сгенерированные на предыдущем шаге эмуляции.

RandomBlockShift – случайно генерируемый идентификатор нейрона, соответствующего нулевому блоку нитей CUDA. Случайный сдвиг необходим для более равномерного выбора «победившего» нейрона среди одновременно испустивших спайк нейронов популяции WTA.

Модель распараллеливания для этой процедуры – один поток соответствует одному синапсу (для возбуждающих синапсов) или одной секции тормозных синапсов или синапсов, индуцирующих пластичность, с равными весами, координата Y двумерной решетки – одному нейрону. Соответственно, **сумма числа возбуждающих синапсов и числа секций тормозных синапсов** **и синапсов, индуцирующих пластичность, одного нейрона не может превышать 1024**.

Каждый возбуждающий синапс имеет битовую очередь пресинаптических спайков размера 30. Для доступа к своей очереди из любого потока используется макрос DflIncomingPulsesExcitatory. После каждого такта очередь сдвигается на 1 позицию (логически – за счет сдвига начала очереди). Вновь приходящие спайки записываются в начало очереди. Соответственно, величина максимальной задержки равна 30 мсек.

Для доступа к остальным компонентам свойств и состояния возбуждающего синапса, зависящих от модели, используется макрос DEXCITATORYSYNAPSESTATECOMPONENT(COMPONENT), которому в качестве аргумента передается индекс компоненты.

GPU процедура, реализующая доставку спайков:

\_\_global\_\_ void kerPropagateSpikes(

const unsigned \*pDActiveInputNodes,

const UNS64 \*pu64DPostsynapticExcitatory,

const UNS64 \*pu64DPostsynapticInhibitory,

const UNS64 \*pu64DPostsynapticReward,

unsigned MaxnPostsynapticExcitatory,

unsigned MaxnPostsynapticInhibitory,

unsigned MaxnPostsynapticReward,

unsigned \*pDExcitatorySynapses,

unsigned \*pDInhibitorySpikes,

unsigned \*pDRewardSpikes,

unsigned ntactSpikeWasEmitted

)

Процедура распространяет спайки по сети.

pDActiveInputNodes – указатель на вектор номеров входов моделируемой секции сети, получивших спайк; положительные номера соответствуют возбуждающим входам, отрицательные - тормозным.

pu64DPostsynapticExcitatory – указатель на массив описаний связей возбуждающих синапсов с входами моделируемой секции сети. Каждое описание состоит из двух полей, упакованных в 64битные целые. Первые 56 бит – это индекс синапса в общем массиве входных спайков возбуждающих синапсов, остальные биты – величина синаптической задержки.

pu64DPostsynapticInhibitory – указатель на массив описаний связей секций тормозных синапсов с входами моделируемой секции сети. Каждое описание состоит из двух полей, упакованных в 64битные целые. Первые 56 бит – это индекс синапса в общем массиве секций тормозных синапсов, остальные биты – величина синаптической задержки.

MaxnPostsynapticReward – указатель на массив описаний связей секций синапсов, индуцирующих пластичность, с входами моделируемой секции сети. Каждое описание состоит из двух полей, упакованных в 64битные целые. Первые 56 бит – это индекс синапса в общем массиве секций синапсов, индуцирующих пластичность, остальные биты – величина синаптической задержки.

MaxnPostsynapticExcitatory – максимальное количество возбуждающих синапсов, связанных с одним входом моделируемой секции сети.

MaxnPostsynapticInhibitory – максимальное количество секций тормозных синапсов, связанных с одним входом моделируемой секции сети.

MaxnPostsynapticReward – максимальное количество секций синапсов, индуцирующих пластичность, связанных с одним входом моделируемой секции сети.

pDExcitatorySynapses – указатель на общий массив входных спайков возбуждающих синапсов.

pDInhibitorySpikes – указатель на общий массив входных спайков секций тормозных синапсов.

pDRewardSpikes – указатель на общий массив входных спайков секций синапсов, индуцирующих пластичность.

ntactSpikeWasEmitted – такт, в который спайки были сгенерированы (это предыдущий такт по отношению к текущему).

Схема распараллеливания этой процедуры: Y координата блока соответствует одному входу, на который в данный такт пришел спайк; вектора индексов возбуждающих синапсов, секций тормозных синапсов или синапсов, индуцирующих пластичность, соответствуют одномерным блокам, занимающим последовательные координаты X в решетке.

## CPU реализация – проект ArNICPU

В CPU варианте процедура пересчета состояния нейронов оформлена как метод NeuLIF::tact. Пересчет разных нейронов выполняется параллельно в нескольких потоках, количество которых устанавливается глобальной переменной NEMULATIONTHREADS. Каждый поток контролируется объектом класса tact\_context, который включает вектор идентификаторов нейронов, находящихся в его ведении. Эти потоки, синхронизируясь барьерами, поочередно вызывают методы tact\_context::tact (который вызывает метод tact у нейронов) и tact\_context::distribute, который распространяет спайки по сети с помощью указателей на пресинаптические нейроны, хранящиеся в структурах синапсов.

# Обработка протокола активности сети

Результатом работы ИНС являются спайки, генерируемые ее нейронами. У ИНС могут быть заранее заданные «выходные» нейроны, чья активность рассматривается как результат, а могут и не быть. В любом случае, чтобы проинтерпретировать этот результат, перевести его из спайковой формы во что-то более понятное и/или употребительное требуются специальные процедуры, иногда сравнимые по вычислительной сложности с самим моделированием ИНС. Это неудивительно – ценная информация, получаемая сетью, часто маскируется сильной шумовой активностью ее тысяч и более нейронов. Процедуры, выявляющие эту ценную информацию, зависят от сути решаемой задачи. Они будут рассмотрены в этом разделе.

Все эти программы работают с протоколом активности сети, имеющим одну из двух форм. Первая форма - это текстовый файл, каждая строка которого соответствует такту моделирования, а позиция в строке – нейрону. Если данный нейрон сгенерировал спайк в данном такте, в соответствующей позиции стоит символ ‘@’, иначе – символ ‘.’. Вторая и более обычная форма – это бинарный файл. Первые 8 байт этого файла – это число нейронов в сети. Далее идут биты генерированных спайков нейронов, упорядоченных по индексам, такт за тактом. Битовые маски, относящиеся к отдельным тактам, имеют выравнивание 8 байт.

Модули обработки и анализа протоколов активности сети оформлены в двух разных видах. Часть из них реализована как коды на языке python. Они находятся в директории postprocessing. Остальные являются консольными приложениями, создающими текстовые файлы отчетов.

## Визуализация мониторинга сети - Results

Результаты мониторинга визуализируются питоновской программой Results.

Вычислительно емкая часть обработки результатов мониторинга реализована в консольном приложении ArNIResults. Эта программа читает файл мониторнига сети (для введенного в книге идентификатора эксперимента) и формирует несколько файлов в формате CSV, которые уже непосредственно показываются программой Results.

Информация по динамике средней частоты генерации спайков в различных секциях сети и размере этих секций получается из файла <id>.secint.csv. Информация о динамике величины изменения весов (что дает возможность судить о приближении сети к равновесию) – из файла <id>.deffw\_sum.csv. Файл <id>.sat\_sup\_dist.csv содержит информацию по динамике распределения числа полностью подавленных (синаптический ресурс отрицателен) и насыщенных (синаптический ресурс > 30) синапсов. Динамика этих распределений отображается с помощью интерактивного слайдера в 2 горизонтальных рядах графиков, соответствующих связям между разными секциями сети. По оси Х отображено число подавленных или насыщенных связей, по оси Y – количество нейронов с таким диапазоном подавленных или насыщенных связей.

# Оптимизация свойств ИНС с помощью генетического алгоритма – проект ArNI\_GA

В настоящее время не существует теоретического метода, позволяющего предсказать общие свойства ИНС, исходя и свойств составляющие ее нейронов, статистики их связей и т.д. Поэтому единственным способом подбора наилучших с точки зрения некоторой решаемой задачи параметров модели нейрона, межнейронных связей и т.п. является численная оптимизация. В условиях, когда неизвестен вид зависимости критерия оптимизации от оптимизируемых параметров, этих параметров много, и они имеют разную природу (числовые, дискретные, булевские) практически единственной надежной техникой оптимизации является генетический алгоритм.

Оптимизация ИНС с помощью генетического алгоритма производится консольным приложением ArNI\_GA. Синтаксис командной строки:

./ArNI\_GA <кол-во используемых карт GPU> <кол-во сетей, одновременно запускаемых на 1 GPU> -(i|o)<имя файла лога>

Если последний параметр начинается с “-o”, результаты работы сохраняются в заданный в этом аргументе файл. Если он начинается с “-i”, работа оптимизатора начинается с последнего сохраненного в файле результата (и дальнейшее сохранение результатов туда продолжается).

Способ расчета критерия оптимизации и набор оптимизируемых параметров задаются в файле, включаемом в ArNI\_GA.cpp (см. файлы \*.inc в проекте ArNI\_GA). Включаемый файл должен содержать следующие компоненты:

1. Определение генов и их начальных распределений. Это должен быть блок строк вида

* CHROMOSOME\_DEFINITION
* {
* INITIALIZE\_GENE(<название гена>, <распределение>),
* …
* };
* распределение – это должен быть вызов какого-то метода объекта rng класса Rand, возвращающего случайное число.

1. Определение зависимостей генов. Это должен быть блок строк вида

GENE\_DEPENDENCIES

{

(GENE\_DEPENDENCY|GENE\_ALTERNATIVE)(<ген>, <зависимый ген>),

…

};

* GENE\_DEPENDENCY означает, что если значение гена равно 0, то значение зависимого гена не используется. GENE\_ALTERNATIVE означает, что если значение гена не равно 0, то значение зависимого гена не используется.

1. Должен быть включен файл Chromosome.hpp, содержащий определение класса хромосом.
2. Должен быть реализован метод bool Chromosome::bNoFatalDefect(void) const, проверяющий, что сочетание генов в данной хромосоме в принципе допустимо. В этом методе можно пользоваться макросом GV(.), возвращающим значение данного гена.
3. Должен быть реализован оператор double Chromosome::operator()(int ind) const, оценивающий данного индивида. Его номер в популяции - ind. В процессе тестирования должен быть сформирован файл конфигурации сети. Чтобы облегчить его создание применяется следующий механизм. Создается шаблон файла конфигурации сети. Это файл конфигурации сети, в котором отдельные конструкции заменяются на выражения вида $(<идентификатор>). Вместо этих выражений производится подстановка значений генов либо других значений. Для реализации этого механизма в файле NNCGA.h создана система макросов. В начале макросом SETNNP устанавливается используемый шаблон. Подстановки делаются макросами SUBMACRO (подставляет значение макроса), SUBGENE (подставляет значение гена как действительного числа), SUBGENEINT (подставляет значение гена как целого числа) и SUBVALUE (подставляет значение второго аргумента в шаблонный идентификатор – первый аргумент). Можно условно исключать целые фрагменты шаблона. Эти фрагменты помечаются парами конструкций <!--идентификатор-->, <!--/идентификатор-->. Фрагменты включаются или исключаются на основе значения гена, переданного как аргумент в макрос CHECKSECTION. Кроме того, фрагмент может быть исключен явным образом с помощью макроса EXCLUDESECTION. После проведения всех подстановок результат может быть сохранен с помощью макроса SAVENNC. Работа с этим механизмом заканчивается с помощью макроса NNCDONE. Далее эмуляция может быть запущена программой ArNIGPU.

* Примеры таких включаемых файлов заданий и шаблонов файлов конфигураций содержатся в пакете (см., например, WTAGPU.inc).
* Реализован кластерный вариант этого алгоритма. Чтобы его запустить на всех узлах кластера, кроме координатора, ArNI\_GA должна быть запущена с единственным аргументом – числом GPU, которые должны быть использованы для счета на данном узле. В рабочей директории координатора должен присутствовать файл ArNIClusterHosts.txt, в каждой строке которого должен быть адрес узла. Если на узле-координаторе ArNI\_GA запускается с заданным количеством GPU, которое больше, чем имеется на этом узле, то координатор размещает задачи по тестированию ИНС на других узлах кластера. В качестве транспортного уровня при этом используется tcp/ip.

# Оптимизация свойств ИНС с помощью координатного спуска – проект ArNI\_CD

Генетический алгоритм – это надежный и устойчивый метод оптимизации. Однако, он очень медленно сходится. Поэтому разумно дополнить его более быстрым методом, применяемым тогда, когда опасность нахождения локального максимума уже невелика – например, после того, как новые поколения генетического алгоритма не дают прогресса. Наиболее быстрые методы, основанные на градиентном спуске, все же и в этом случае оказываются неприменимыми, как в силу технической невозможности вычисления производных критерия оптимизации по дискретным параметрам, так и из-за относительно большого вклада случайных факторов – таких как детальная конфигурация связей в сети. Поэтому был выбран более медленный, но более устойчивый метод координатного спуска.

Оптимизация ИНС с помощью генетического алгоритма производится консольным приложением ArNI\_CD. Синтаксис командной строки:

./ArNI\_CD <кол-во используемых карт GPU> <кол-во сетей, одновременно запускаемых на 1 GPU> <имя xml-файла начала поиска>

Последний параметр – это имя XML файла, описывающего стартовую точку оптимизации. Это XML файл с основным разделом opt. Каждый раздел верхнего уровня соответствует оптимизируемому параметру с таким же именем. Его обязательный атрибут start содержит начальное значение параметра. Кроме того, он может содержать разделы val, описывающие возможные дискретные значения параметра. Если эти разделы есть, start равен номеру этого раздела в порядке их следования (начиная с 0). Также, раздел верхнего уровня может содержать разделы dependent. В этих разделах перечисляются параметры, зависящие от данного параметра (см. конструкцию GENE\_DEPENDENCY из предыдущей главы).

Способ расчета критерия оптимизации и набор оптимизируемых параметров задаются в файле, включаемом в ArNI\_CD.cpp (см. файлы \*.inc в проекте ArNI\_CD). Включаемый файл должен содержать функцию int evaluate(const vector<int> &v\_point, const vector<dimension> &vdim\_, bool &bTerminate), вычисляющую значение критерия оптимизации для набора параметров v\_point (позиция параметра в этом векторе равна порядку соответствующего раздела верхнего уровня в файле, задающем стартовую точку поиска). Содержимое этой функции в большой степени аналогично double Chromosome::operator()(int ind) const из предыдущего раздела, но вместо макросов SUBGENE и SUBGENEINT, там определяются макросы SUBDIM и SUBDIMINT.

* В качестве примера можно использовать seq.cd.inc.
* Реализован кластерный вариант этого алгоритма, полностью аналогичный по логике использования кластерному варианту ArNI\_GA.

# Структурный генетический алгоритм - проект ArNIGAstruct

Обычный генетический алгоритм оптимизирует вектора (или, в нашем случае – списки) значений параметров. Это означает что мы должны заранее знать структуру сети – и с помощью него сможем подобрать наилучшие параметры для этой структуры. Но, скорее всего, оптимальная для решения данной задачи структура сети тоже не известна. Для решения этой проблемы применяется структурный генетический алгоритм. В этом алгоритме скрещиванию и мутациям подвергаются не вектора значений признаков, а сами нейросети. Например, скрещивание двух ИНС может быть произведено двумя способами. В первом пространственная структура обеих сетей случайным образом рассекается пополам и две половинки от разных сетей объединяются в одну новую сеть после соответствующего пространственного преобразования и восстановления разорванных связей в соответствии с логикой популяций пресинаптических нейронов. Во втором из обеих сетей случайным образом выбрасывается половина нейронов и то, что остается, пространственно совмещается, опять же, с восстановлением разорванных связей. Мутация реализуется как случайный выбор нейронной популяции в сети и случайное изменение случайно выбранного параметра этой популяции. В остальном логика генетического алгоритма остается такой же.

Оптимизация ИНС с помощью генетического алгоритма производится консольным приложением ArNIGAstruct. Синтаксис командной строки:

./ArNIGAstruct <кол-во используемых карт GPU> <кол-во сетей, одновременно запускаемых на 1 GPU> <имя log-файла ArNI\_GA> <имя формируемого log-файла>

Предполагается, что структурный генетический алгоритм запускается после и на основе обычного генетического алгоритма (после всего 2-3 его поколений). Он использует финальные сохраненные сети, оставшиеся после запуска ArNI\_GA, и сам формирует эволюционирующие популяции таких сетей. Операции скрещивания и мутации реализуются отдельными программами – NNCrossingover и NNMutation.

# Процедура верификации модели нейрона.

Данный пакет создан, в том числе, для того, чтобы тестировать новые модели нейронов и синапсов. Предполагается, что реализовываться они будут на GPU. Программирование GPU является до сих пор существенно более сложным, чем разработка программ для CPU, и в особенности сложной является отладка. Поэтому очень важен вопрос верификации корректности реализации модели. Для такой верификации предлагается следующий протокол.

1. Любая модель должна быть реализована как на GPU, так и на CPU – т.е. оформлена в виде собранных программ ArNIGPU и ArNICPU.
2. В качестве тестовой берется конфигурация сети, существенным образом использующая все основные свойства новой модели.
3. Используя один и тот же конфигурационный файл и одинаковые командные аргументы, получаем протоколы активности сети с помощью программ ArNIGPU и ArNICPU.
4. С помощью утилиты intdiff сравниваем оба протокола, указав 2 файла протокола в качестве ее аргументов. Как уже говорилось, моделирование ИНС с идентичными параметрами на CPU и GPU не всегда дает в точности одинаковые протоколы активности, однако общие характеристики активности должны иметь примерно одинаковую динамику в обоих случаях. Утилита intdiff выводит количество спайков в каждом такте обоих протоколов через запятую в стандартный вывод. Кроме того, она выводит номер такта и номер нейрона у первого встретившегося отличия в заданных протоколах. Если различий не обнаружено, верификация считается пройденной. В противном случае, выполняется следующий пункт.
5. Повторяется запуск программ ArNIGPU и ArNICPU, но с использованием опции –V. В результате образуются два файла потактовых сигнатур состояния сети. Если они одинаковы, повторяем п. 4. Если нет – находим первый такт с отличающимися сигнатурами.
6. Повторяется запуск программ ArNIGPU и ArNICPU, но с использованием опции –s указанием такта первого отличия в сигнатурах. В результате этого получаются два файла состояния сети. С помощью утилиты бинарного сравнения файлов находим позицию первого отличия. Если отличий нет, повторяем п. 5. В противном случае, выполняется следующий пункт.
7. С помощью отладчика выясняем, сериализация чего показывает отличие. С помощью отладчика и отладочной выдачи устанавливаем причины этого отличия. Если выясняется, что это происходит из-за ошибки, исправляем ее и повторяем п. 6. Отличие может объясняться неустранимым различием между реализациями на CPU и GPU, либо неустранимым недетерминизмом. В этом случае переходим к следующему пункту.
8. Сравниваем интенсивности генерации спайков, выведенные утилитой intdiff. Если отличия несущественны, констатируем верификацию. В противном случае привлекаем дополнительную технику нахождения отличий и выявления их причин.