ARNI-X: A SIMULATOR FOR SPIKING NEURAL NETWORKS

12/7/2024

Version 2.1

By Mikhail Kiselev

ArNI-X: A simulator for spiking neural networks

VERSION 2.1

CONTENTS

Copyright iv
Introduction 1
General information 1
Installation 1
Running SNN Emulation using ArNI-X 1
Tutorials 2
Single Neuron 2
Liquid State Machine 3
Unsupervised Learning 5
Neuron Model 11
Synaptic Plasticity Rules 12
Hebbian Plasticity 13
2-Factor Dopamine Plasticity 15
3-Factor Dopamine Plasticity 15
Synaptic Resource Renormalization 15
Neuron Stability 16
Network Structure Description 17
Overall Structure of NNC Files 17

CONTENTS

RECEPTORS – Description of Input Nodes ……..……..... *18*

NETWORK – Description of SNN Structure ……..……..... *19*

Emulator Command Line Arguments ……..... **24**

Example of Monitoring File Processing Using a Python Script ……..... **25**

Copyright

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, there 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, no programming is 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 three simple examples in the Tutorials. **It is strongly recommended to read the Tutorials first.** 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

No special installation procedure is required – just the corresponding archive file unpacking with conserved directory structure.

This system uses boost libraries. If boost is not installed then the libraries in the `boost` folder (in the distribution package for Linux) should be copied to some place from which the OS loads shared libraries.

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 dynamic library fromFile from the distribution package should be copied to this 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 input type is "none" (the attribute of the args node), 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 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 the 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.

It should be noted that in the present version of ArNI-X for Linux, a small bug exists, which sometimes requires execution of the command `reset` in the terminal after running the emulator.

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.

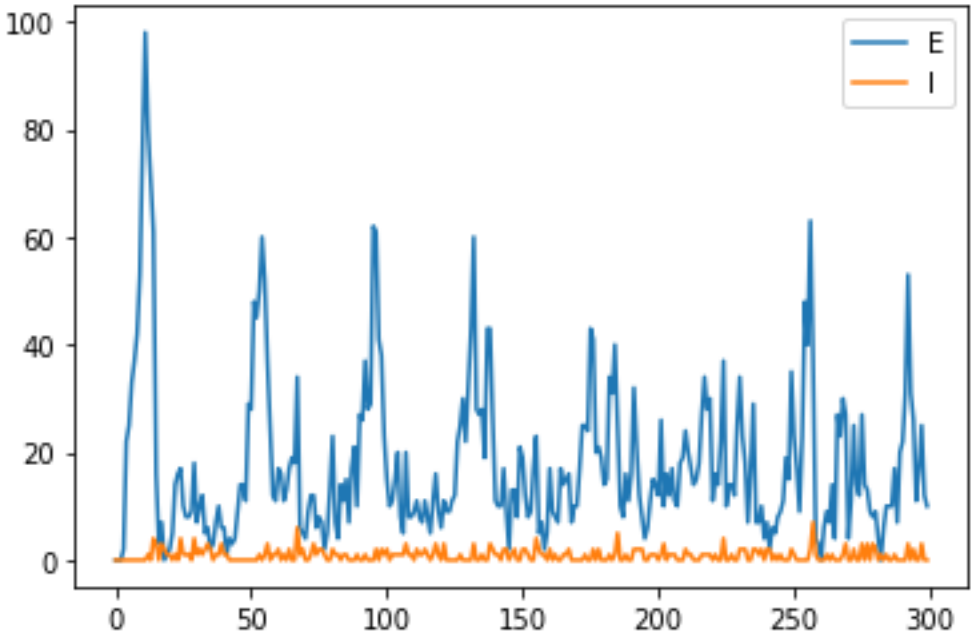
However, this time, excitatory neuron model is more complex than simple LIF. Threshold potential of these neurons is not constant. Every time the neuron fires it is incremented by 1 (see the `threshold_inc` parameter); after that it linearly drops to its rest value **8.531**. The speed of this decrease 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 (it may require installation of some additional python packages). Here it is:

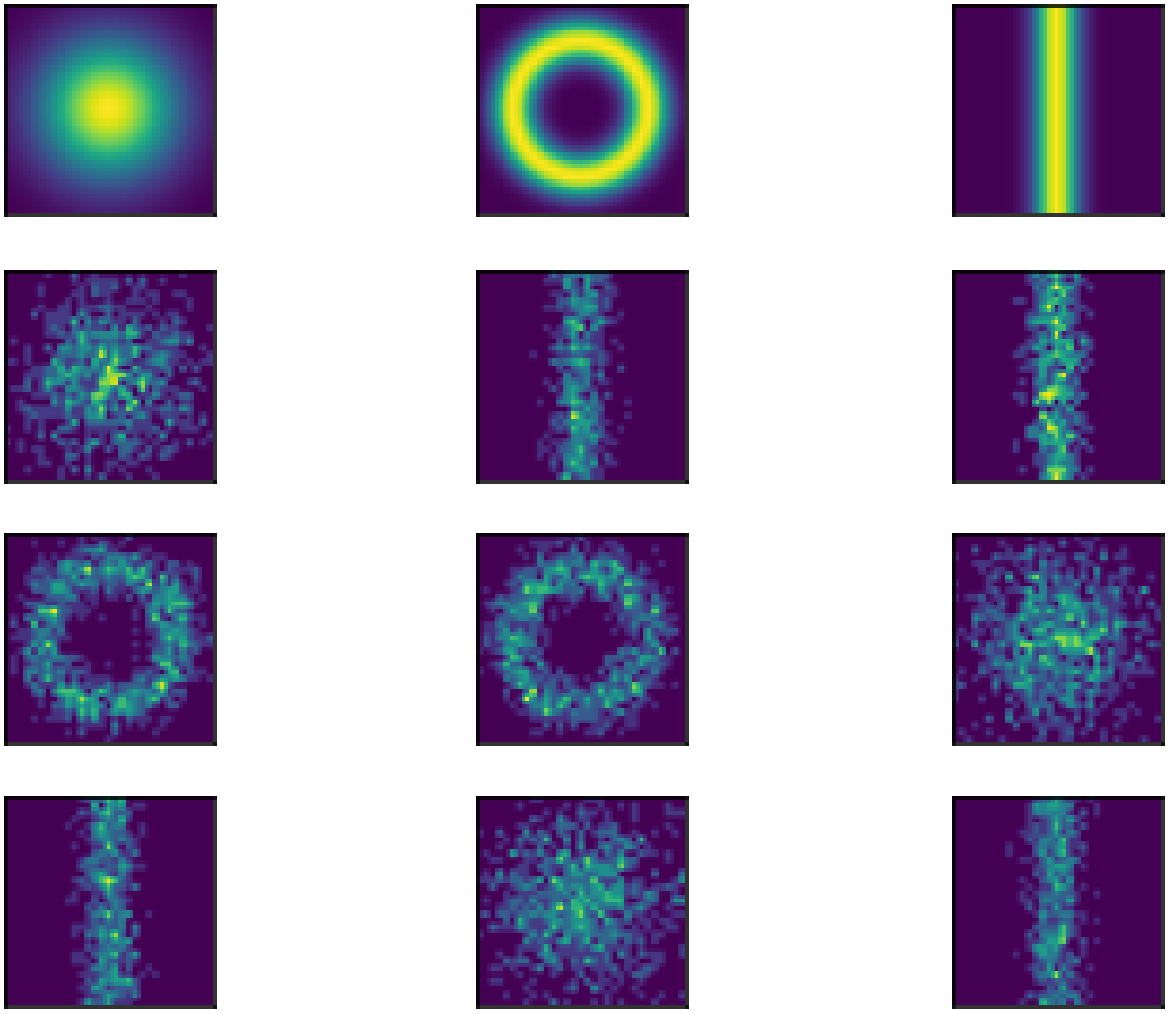


We see that the E population demonstrates non-trivial rhythmic behavior with the frequency about 20 Hz.

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.

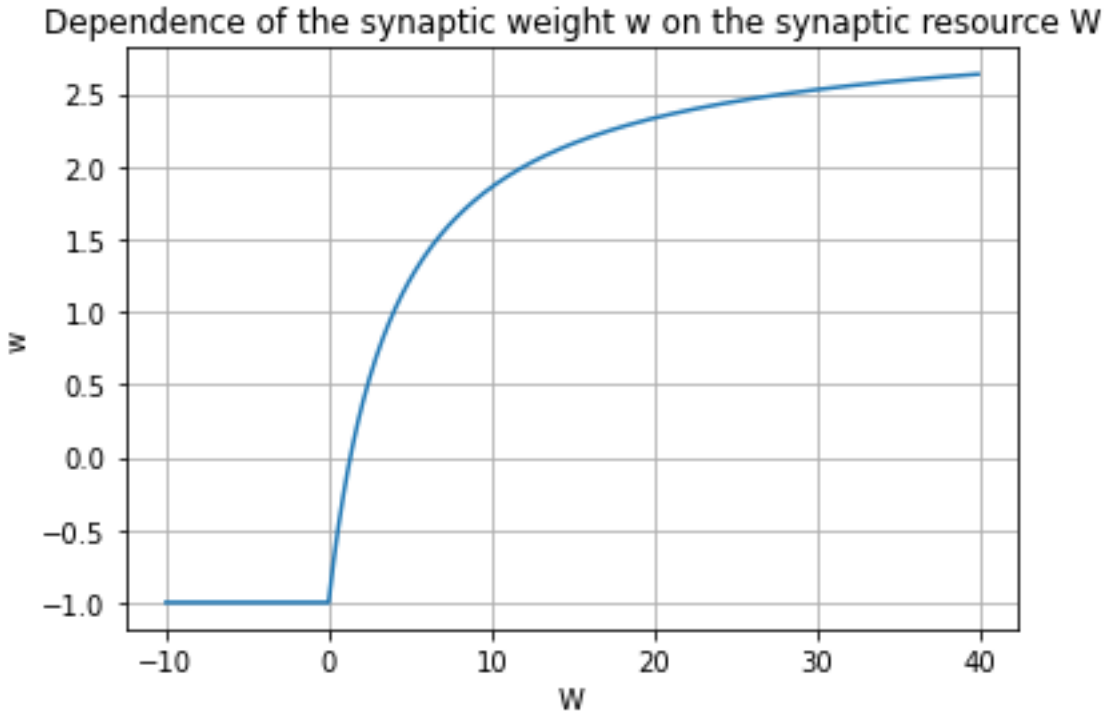
The neurons are connected to the input nodes but these are not "all-to-all" connections. In order to learn to recognize different patterns, the neurons should be sufficiently different from the very beginning. One way to get it is to connect neurons with different subsets of input nodes (but these subsets 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

$$w = w_{min} + \frac{(w_{max} - w_{min})\max(W, 0)}{w_{max} - w_{min} + \max(W, 0)}.$$

In this model, the weight values lay inside the range $[w_{min}, w_{max})$ - while W runs from -∞ to +∞, w runs from w_{min} to w_{max} (see the picture below, where w_{min} = -1, w_{max} = 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 w_{min} to w_{max} 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 d_H depends on the basic neuron plasticity value $\overline{d_H}$ 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

$$d_H = \overline{d_H}\min(2^{-s}, 1).$$

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 t_D 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). t_D 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

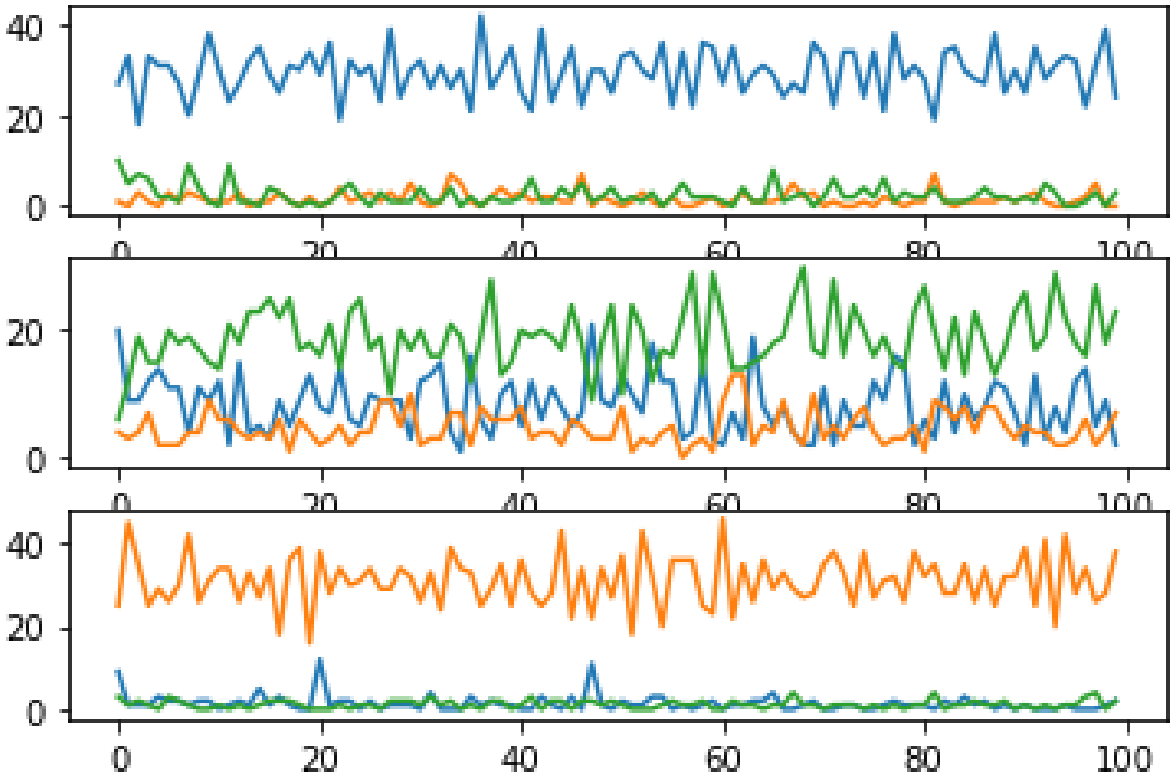
```
ArNICPU ..\Tutorials -e3 -Pt -v1 -F5000
```

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 `-F5000` tells that it will be stored every 5000 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`.

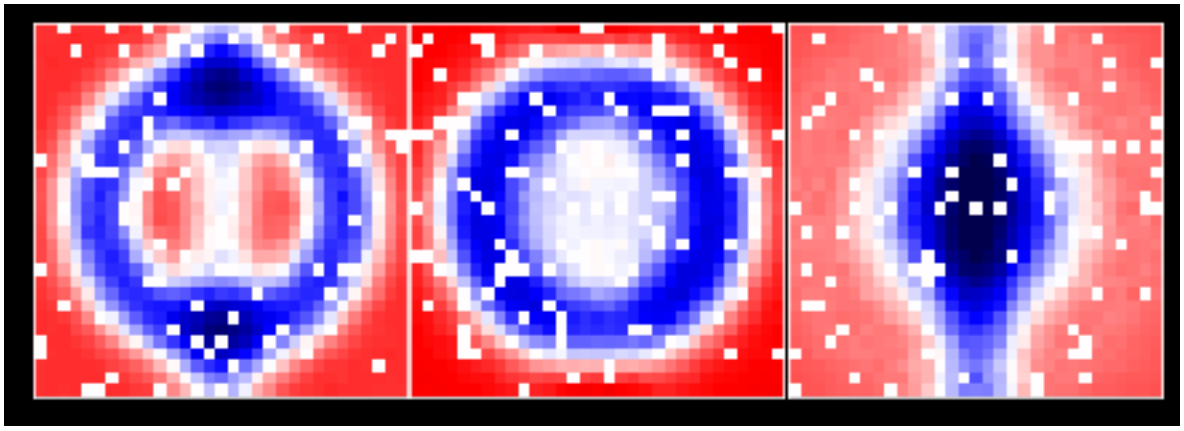
Now let us evaluate the results of unsupervised learning. We break the whole emulation period to 300 msec intervals. In each interval, we count firings of all neurons when each of 3 patterns are presented. In case of successful learning, every neuron should demonstrate systematically higher firing frequency 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 - the final values of plastic weights contained in the monitoring file `monitoring.3.csv`. It is a text comma-separated-values (CSV) file which has complex structure changing from version to version. It is parsed by the python script file `ClusteringWeights.py` (see also the final section of this manual) which 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 $u_{THR}(t)$. Dynamics of these values are defined by the equations

$$\begin{cases} \dfrac{du}{dt} = -\dfrac{u}{\tau_v} + \sum\limits_{i,j} w_i \delta\left(t - t_{ij}\right) \\ \dfrac{du_{THR}}{dt} = -a\ \mathrm{sgn}(u_{THR} - H) + \sum\limits_{k} \hat{T}\delta(t - \hat{t}_k) \end{cases}$$

and the conditions that u is hard limited from below by the value u_{MIN} and that if u exceeds u_{THR} 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 $u \to 0$ and $u_{THR} \to H = 8.531$ (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 u_{THR} to its base value H; w_i - the weight of i-th synapse; t_{ij} - the time moment when i-th synapse received j-th spike; $\hat{T}$ – u_{THR} is incremented by this value when the neuron fires at the moment $\hat{t}_k$. 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 $\tau_M < \hat{T}/a$, then at the moment of the timer reset, u_{THR} 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.

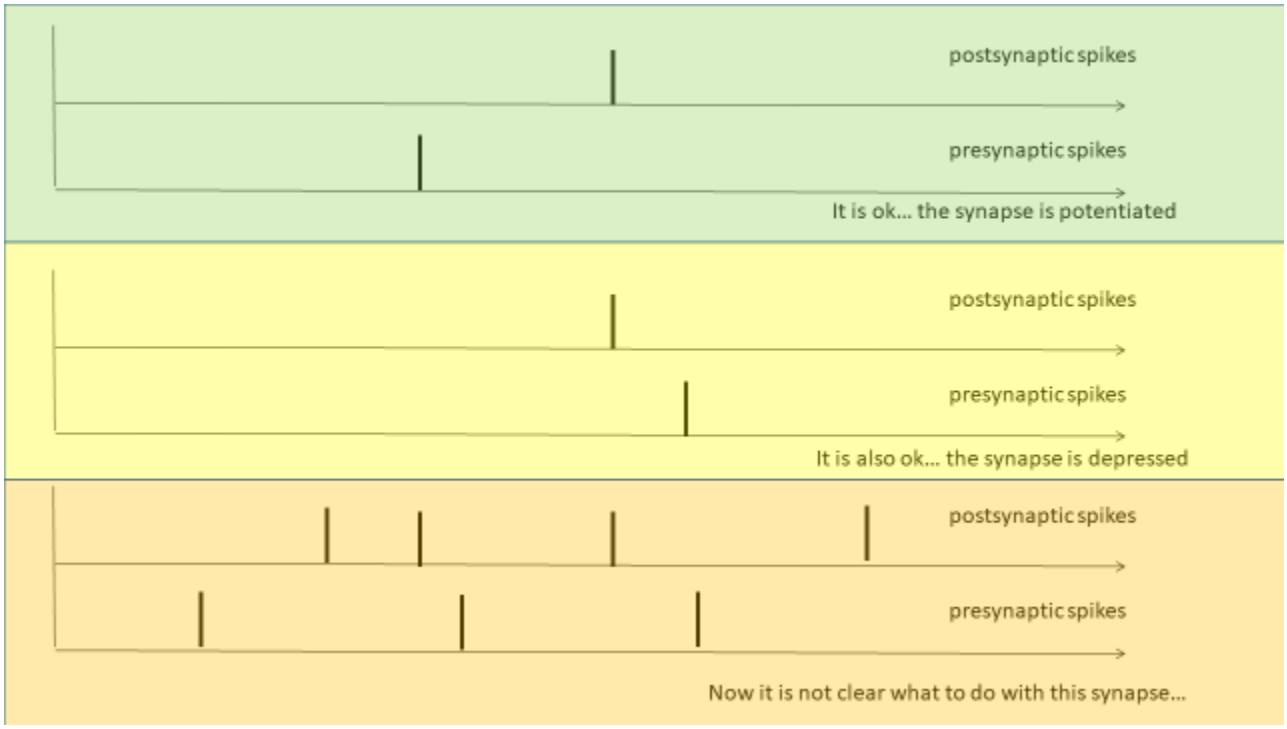
SYNAPTIC PLASTICITY RULES

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 ISI_{max} (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 ISI_{max} before the first spike in TSS;
2. Inter-spike intervals for all neighboring spikes in TSS are not greater than ISI_{max};
3. There are no spikes during time ISI_{max} 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 T_H 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 T_H should be few times greater than τ_v – we selected $T_H = 3\tau_v$.
3. All synaptic resources are changed by the same value d_H independently of exact timing of presynaptic spikes.

It should be added that d_H 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 T_D 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 $R \leq 0$.

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 T_D 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 $T_H = 3\tau_v$ 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 d_H in the Hebbian plasticity rule depends on s this way:

$$d_H = \overline{d_H}\min(2^{-s}, 1).$$

Here, $\overline{d_H}$ is the basic level of Hebbian plasticity.

For dopamine plasticity, the resource change is $D\min(2^{-s}, 1)$, 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 $r\overline{d_H}$, 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	$rR\max\left(2 - \frac{\lvert t_{TSS} - ISI_{max}\rvert}{ISI_{max}}, -1\right)$, where t_{TSS} – 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	$2rR$
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 – $2rR$.

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 `NETWORK` nodes. The former describe input nodes (there may be several input node sections sending signals with different semantics), the latter – the components of 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.

NNC files can include contents of other NNC files. To include NNC file, its name should be written in a separate line prepended by back slash, like this:

```
\included.nnc
```

In this case, all `RECEPTORS` and `NETWORK` nodes from the included file are inserted. This feature allows to construct complex network from simpler components.

`RECEPTORS` – Description of Input Nodes

A `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.

A `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 from a file and add Poissonian noise.

An `Implementation` node should include only one node with the tag `args` with the attribute `type`. This attribute can take the following values:

- `none` – No external input data (only Poissonian noise).
- `text` – The input spikes are read from the 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` – The input spikes are read from the binary file. The 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.
- `image` – A special kind of binary input signal file – a set of monochrome images. See the `Special` tag below.

The `args` 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 is normal (the `mode` node is not present), 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` – the signal generation regime. 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. If it is not specified, stdin is used.
- `Special` – if present, this node specifies a special format of the input signal file (determined by the `type` attribute). At present, this node can be included only one for `type` equal to `image`. In this case, the input signal file contains monochrome images stored tightly image by image, row by row. 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 described in the sub-node `props`.

The following parameters can be specified as the sub-nodes of `props`:

- `bursting_period` – memory spike train period τ_M (msec)**.** By default, neurons have no capability of constant firing (and therefore, their $\tau_M = 0$). However, if the `memory` property is present then the neuron of the population described have this ability, and the default value of τ_M is 9 msec.
- `chartime` **–** the membrane leakage time constant τ_v (msec). The value of this node may be `INFINITY`. In this case, there is no membrane potential leakage. The default value is 1. Since every emulation step, the membrane potential is multiplied by $1 - 1 / \tau_v$, $\tau_v = 1$ means that by default the membrane potential is zeroed before every emulation step.
- `dopamine_plasticity_time` – T_D (msec).
- `maxTSSISI` – ISI_{max} (msec) – the maximum inter-spike interval in TSS. By default, this value is 0 – no TSS, only single postsynaptic spikes are taken into account.
- `maxweight` – the maximum value of the plastic synapse weight w_{max}.
- `memory` – the approximate duration of neuron memory implemented as a sequence of periodical firings (msec) – see the `bursting_period` parameter. `memory` can have the following values:
  - a number. This is memory duration in msec.
  - `INFINITY`. In this case, the only possible way to stop the periodical firing is inhibition from some other neuron.
  - `UNI(`<min>`,`<max>`)`. In this case, neuron's memory duration value is a uniformly distributed random number from the range (min, max).
  - `LU(`<min>`,`<max>`)`. In this case, natural logarithm of neuron's memory duration value is a uniformly distributed random number from the range (ln(min), ln(max)).
- `minpotential`– the minimum value of the membrane potential u_{MIN}. The default value is very high negative number (no lower limit for the membrane potential).
- `minweight` – the minimum value of the plastic synapse weight w_{min}. The default value is 0.
- `n` – the number of neurons in the population. It is the only mandatory parameter.
- `nsilentsynapses` – number of (imaginary) additional plastic synapses not connected to any spike source. They are used in the synaptic resource renormalization procedure as a reservoir for

excessive synaptic resource (or as a source of synaptic resource deployed on the working plastic synapses). The default value is 0 that corresponds to exactly preserved neuron's total synaptic resource. The value -1 means that the total synaptic resource constancy is not maintained.

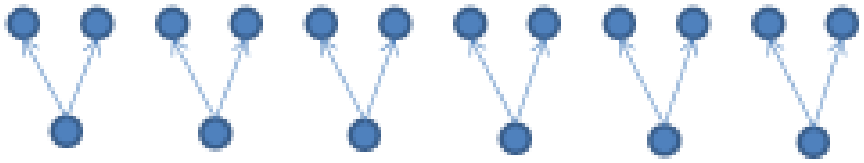
- `stability_resource_change_ratio` – the coefficient of proportionality *r* between the neuron stability change and synaptic resource change for the case of zero stability.
- `Structure` – the population geometrical structure. If it is defined, it can determine the pre/post connection probabilities and the synaptic delays. This node should have one obligatory attribute `type` and optional attributes. The attribute `type` determines the type of structural organization of the population. At present, only one type is implemented – *columnar structure*, corresponding to the value of `type` equal to `0`. For this structure type, one more attribute `dimension` should be included in the `Structure` node and be equal to an integer number interpreted as the number of subpopulations ("columns") inside the population. In this case, the population is broken to the set of non-intersecting groups of the equal size. This structure is taken into account when inter-neuron connections are created (see below).
- `three_factor_plasticity` – presence of this tag means that the neurons in this population obeys 3-factor dopamine plasticity rule (2-factor dopamine plasticity rule is default).
- `threshold_decay_period` – the time t_Θ necessary for the threshold potential to reach its basic value after a single stand-alone firing (msec). $a = \hat{T}/t_\Theta$.
- `threshold_inc` – the threshold potential increment $\hat{T}$.
- `weight_inc` – the basic level of Hebbian plasticity $\overline{d_H}$.

`Link` – Projection Property Definition

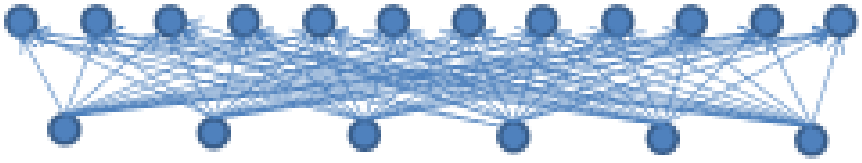
Projection is a set of connections between neurons. One projection can include connections between two neuron populations (or between a population and itself). The connections belonging to one projection have similar properties, parameters either equal or taken from the same random distribution. The XML node `Link` describing a projection has two obligatory attributes and two optional ones. The two obligatory attributes, `from` and `to`, specify the names of the populations connected. `from` may be the name of a population or an input node section. The two optional attributes are:

- `type` – connection type (see the description of synapse types above). It may be `plastic`, `reward` or `gating`. If there is no `type` attribute, the synapses belonging to the projection are fixed.
- `policy` – connection policy. It may be one of the following (see the picture below):
  - `aligned`. If the population are of the same size, then it is one-to-one projection. The neurons with the same index inside their populations are connected. If the presynaptic population is smaller then every neuron from it is connected to *n* neurons where *n* is integer part of the ratio of the postsynaptic population size and the presynaptic population size. Again, all neurons are selected in order of their indices inside their populations. If the postsynaptic population size is not a multiple of the presynaptic population size, some postsynaptic population neurons will stay unconnected. If the postsynaptic population is smaller, then the situation is mirror-symmetric.
  - `all-to-all`. All-to-all connections (but reflexive connections are prohibited!).

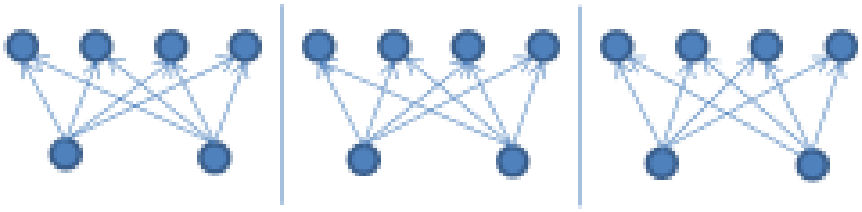
- `all-to-all-sections`. This connection policy is only allowed if both connected populations have the same columnar structure (see the tag `Structure` inside `Section`). In this case, all-to-all connections but inside the sections with the same indices are created.
- `exclusive`. If both connected populations have the same columnar structure then these are all-to-all connections excluding connections between the neurons in columns with the same index. Otherwise, these are all-to-all connections excluding connections between the neurons with the same index inside their populations. For example, winner-takes-all blocking lateral connections can be made using this policy.
- `exclusive-sections`. This connection policy is only allowed if both connected populations have the same columnar structure and the same size. Inside the columns with the same index, the all-to-all connections are created except the connections between the neurons with the same index inside their populations.



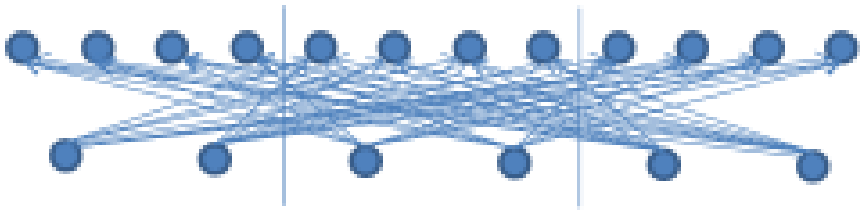
aligned



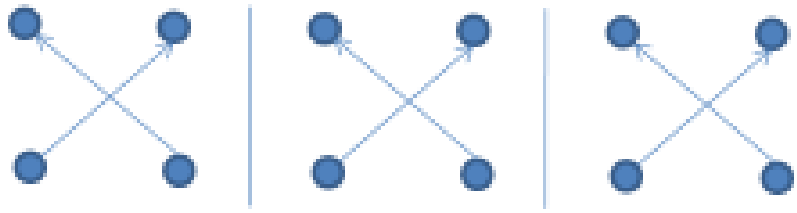
all-to-all



all-to-all-sections



exclusive



exclusive-sections

If the `policy` attribute is not defined, the connections between the populations are random.

Inside the `Link` node, there should be nodes describing the projection properties:

- `Delay` – synaptic delay distribution (in msec). This node should have the attribute `type`, which can take one of two values – `uni` or `ln`. The former corresponds to uniform distribution, the latter – to log-normal distribution. In the first case the node `Delay` should include the nodes `min` and `max`, defining the range of delays. If their values are equal then all delays are set to this number. In the case of log-normal distribution, there should be the sub-nodes `mean` (with the numeric value M) and `stddev` (with the numeric value d). The random delays are generated using the formula $M\exp(\mathbb{N}(d))$, where $\mathbb{N}(d)$ is a normally distributed random value with the center in 0 and the standard deviation d. The random delay is hard limited from above by the value 30 msec – the longest possible delay in ArNI-X.
- `IniResource` – initial synaptic resource distribution. This node should have the attribute `type`, which can take one of two values – `uni` or `dis`. `uni` corresponds to uniform distribution. In this case, the node `IniResource` should include the nodes `min` and `max`, defining the range of initial resource values. If their values are equal then all initial resources are set to this number. `dis` means discrete distribution. If `type="dis"`, the node `IniResource` should contain one node `default` and, optionally, several nodes `value`. Every `value` node should have the attributes `v` and `share`. The attribute `share` should be a number from the range (0, 1). It is the probability that the initial value of a synaptic resource equals to the value of the respective attribute `v`. The sum of all shares should not exceed 1. If the sum is less than 1, then a synaptic resource takes the value `default` with the probability equal to the difference between 1 and this sum. By default, synaptic resources are initialized by 0.
- `maxnpre` – the maximum number of synapses belonging to this projection per neuron. This property can be set only for the default connection policy. The default value is very great (no limits on synapse counts).
- `probability`– the probability that two given neurons from the populations `from` and `to` will be connected. This property must be set only in the case of the default connection policy.
- `weight` – the synaptic weight. This property must be set for all projection types except `plastic` (for plastic connections, the initial resource is specified instead of weight).

EMULATOR COMMAND LINE ARGUMENTS

The emulator command line has the syntax `ArNI(C|G)PU` <ExperimentSeriesDirectory> <Options>*. `ArNICPU` preforms the emulation on CPU, `ArNIGPU` – on GPU. ExperimentSeriesDirectory should contain all NNC files belonging to one emulation experiment series. The Options are following:

-`C`(<CardNo>[,<CardNo>]|`N`<NCores>). The first form specifies GPU ids used for emulation (e.g. -`C0,2`). The second – number of CPU cores used (-`CN10`). By default, all available cards/cores are used.

-e<ExperimentNo>. The configuration file <ExperimentNo>.nnc is used. It is the only mandatory option.

-F<MonitoringPeriod>. Sets the periodicity of network status saving (msec). The default value is 200000.

-f<NetworkFixingIteration>. If present this option sets the emulation iteration number after which the network becomes non-plastic – all its synaptic weights values are fixed.

-P(b|t|l)[<iterbeg-iterend>]. This option controls network activity recording. The letters b, t or l determine the recording format. The text format (t) was described in Tutorials. In case of binary format (b) the network neuron firings are stored as bit masks in the file spikes.< ExperimentNo>.bin. The first 4 bytes of this file is the neuron count in the network. After that the bit masks go sequentially with 8-byte alignment. l corresponds to the list format (spikes.< ExperimentNo>.lst). In this case, the resulting file contains one line per neuron. The *i*-th line contains iteration numbers of all *i*-th neuron firings consecutively in the form of comma separated values. By default, the recording is carried out during the whole emulation, but its period can be specified explicitly.

-R[S][<Seed>]. Using this option, the emulation can be randomized (due to random resetting of the internal random number generator). Using -RS, the input spike sources can be also randomized. If Seed is specified, this randomization is deterministic.

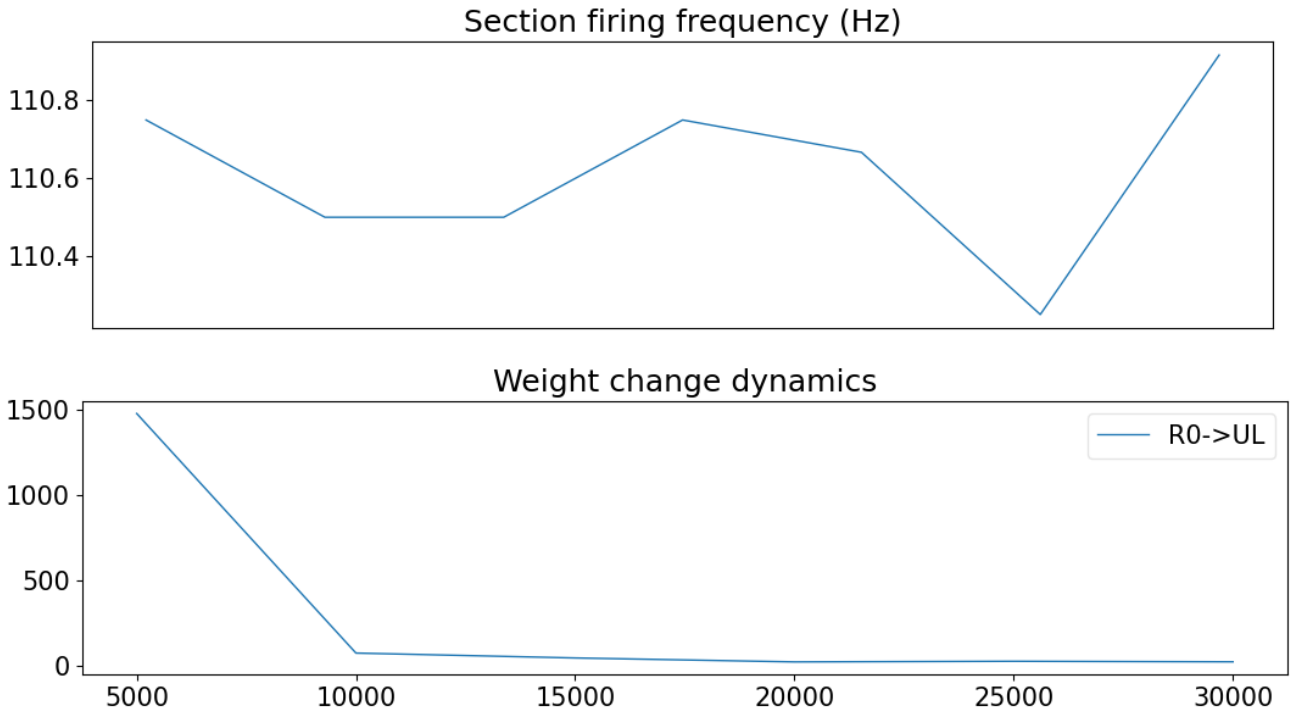
-r. This option saves input node activity recording in the same format as for network activity. The base name of the resulting file is receptor_spikes.

-T<TerminationIterationNo>. This option allows to stop the emulation at the iteration number specified.

-v(0|1|2). This option sets the emulation output level. In the case of the zero value (default), no output is produced (except, maybe, the activity record file – see the option P). Value 1 adds the monitoring file (monitoring.< ExperimentNo>.csv). Value 2 adds the verbose log file <ExperimentNo>.log containing records of all events (firings, synaptic plasticity acts etc.). It may be really huge.

EXAMPLE OF MONITORING FILE PROCESSING USING A PYTHON SCRIPT

As an example of a python script parsing a monitoring file, the file Results.py is included in this package (in the directory Workplace). This script reads the monitoring file monitoring.3.csv from Tutorial 3 and draws the time courses of firing frequency and synaptic weight changes.


Section firing frequency (Hz)
110.8
110.6
110.4
Weight change dynamics
1500
1000
500
0
R0->UL
5000
10000
15000
20000
25000
30000