ARNI-X: A SIMULATOR FOR SPIKING NEURAL NETWORKS

6/27/2025

Version 3.0

By Mikhail Kiselev

ArNI-X: A simulator for spiking neural networks

VERSION 3.0

CONTENTS

Copyright... iii

Introduction... 1

General information.. 1

Installation 1

Running SNN Emulation using ArNI-X 1

Tutorials 2

Single Neuron 2

Liquid State Machine 3

Supervised Learning – Written Digit Classification 5

Neuron Model 16

Synaptic Plasticity Rules 18

Hebbian Plasticity 18

Dopamine Plasticity 20

Synaptic Resource Renormalization 21

Neuron Stability 21

Lower Excitability of Trained Neurons.. 23

Synaptic Weight Quantization.. 23

Network Structure Description 23

Overall Structure of NNC Files 23

CONTENTS

RECEPTORS – Description of Input Nodes ……………….……….............................. 24

NETWORK – Description of SNN Structure ……………….……….............................. 27

Simulator Command Line Arguments ……………….……………………………… 31

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 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 the 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 Simulation 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 simulation results are saved in this directory. It is assumed that computational experiments with SNNs go in series such that every individual simulation run has the series name and the numeric id in the series. In order to run a simulation, the user should specify the SNN structure using a special file <experiment_numeric_id>`.nnc`. All nnc files related to one experiment series should be in separate directory. The dynamic libraries `fromFile` and `StateClassifier` from the distribution package should be copied into this directory. The nnc files contain SNN structure definition in the XML language; the format of these definitions is described in the subsequent sections of this manual. The simulation is performed by the executable files `ArNICPU` (for CPU) or `ArNIGPU` (for GPU). The experiment series name and the experiment id are specified as command line arguments (see below).

The simulation duration can be 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 simulator `ArNICPU`. We recommend launching the simulator 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 the slash should be used instead of the 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 simulation 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 two input spike source types – `fromFile` and `StateClassifier` (see Tutorial #3). The `fromFile` 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. Henceforth, we will take each emulation step equal to 1 msec. Thus, the mean noise frequency for each input network node is 300 Hz. The `history_length` node specifies the emulation duration. It is equal to 1000 (=1 sec).

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 decremented by H. 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 the simulation 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 299 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.

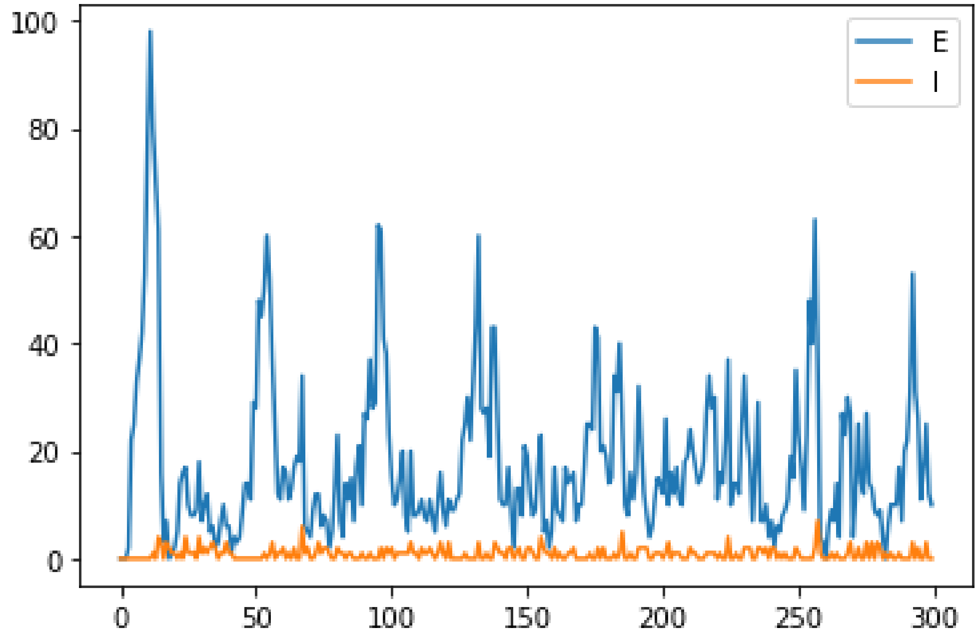
However, this time, excitatory neuron model is more complex than simple LIF. Threshold potential of these neurons is not constant. Every time the neuron fires it is incremented by 1 (see the `threshold_inc` parameter); after that it linearly drops to its rest value **8.531**. The speed of this decrease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

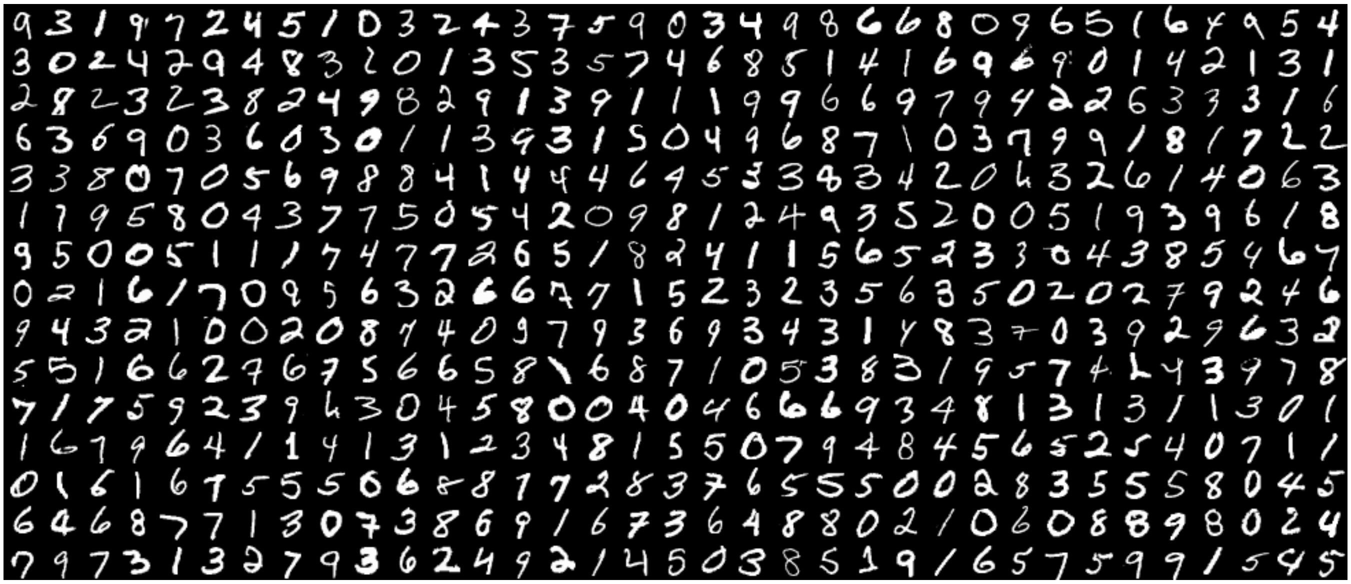


We see that the E population demonstrates non-trivial rhythmic behavior with the frequency about 20 Hz.

Supervised Learning – Written Digit Classification

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 supervised learning. In this learning regime, several different classes of objects are presented to the network in the form of spike streams from its input nodes. After presentation of each object, a special separate input node emits spike indicating the class which the presented object belongs to. During the learning process, network's synaptic weights should be adjusted to form specific reaction of network to objects from different classes – there should be neurons in the network which respond by firing to presentation of objects from one specific class.

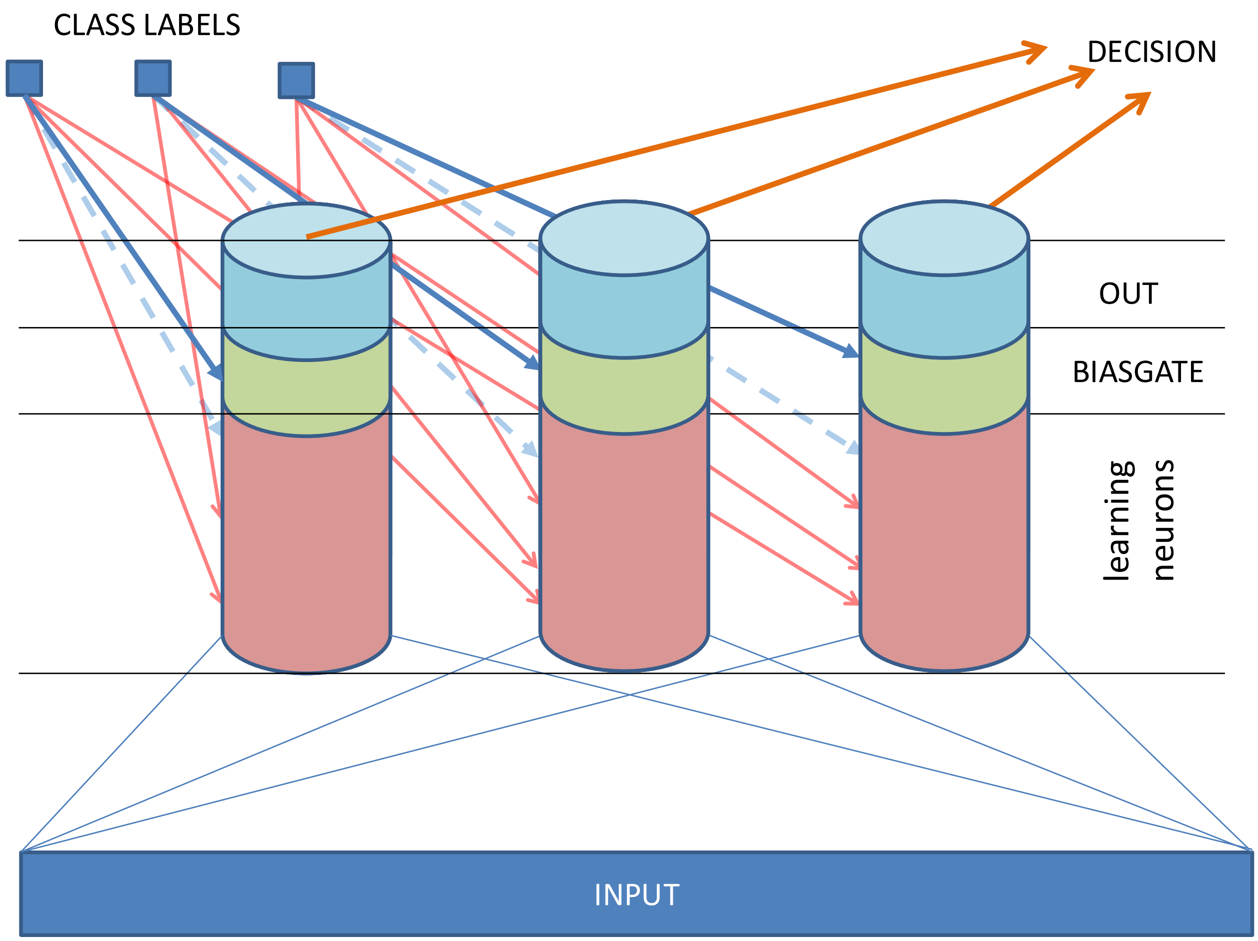
In this tutorial, we use the well-known image set called MNIST. It contains 60000 small monochrome images of hand written digits (0 – 9) used for learning and 10000 digit images used for testing. Each image has size 28×28 pixels, therefore, the network has 784 input nodes – one input node corresponds 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 1 for the brightest pixels. Besides that, the network has 10 additional input nodes serving for labelling the image presented. Each label node emits spike just after presentation of an image from the corresponding class. Here are examples of these images:



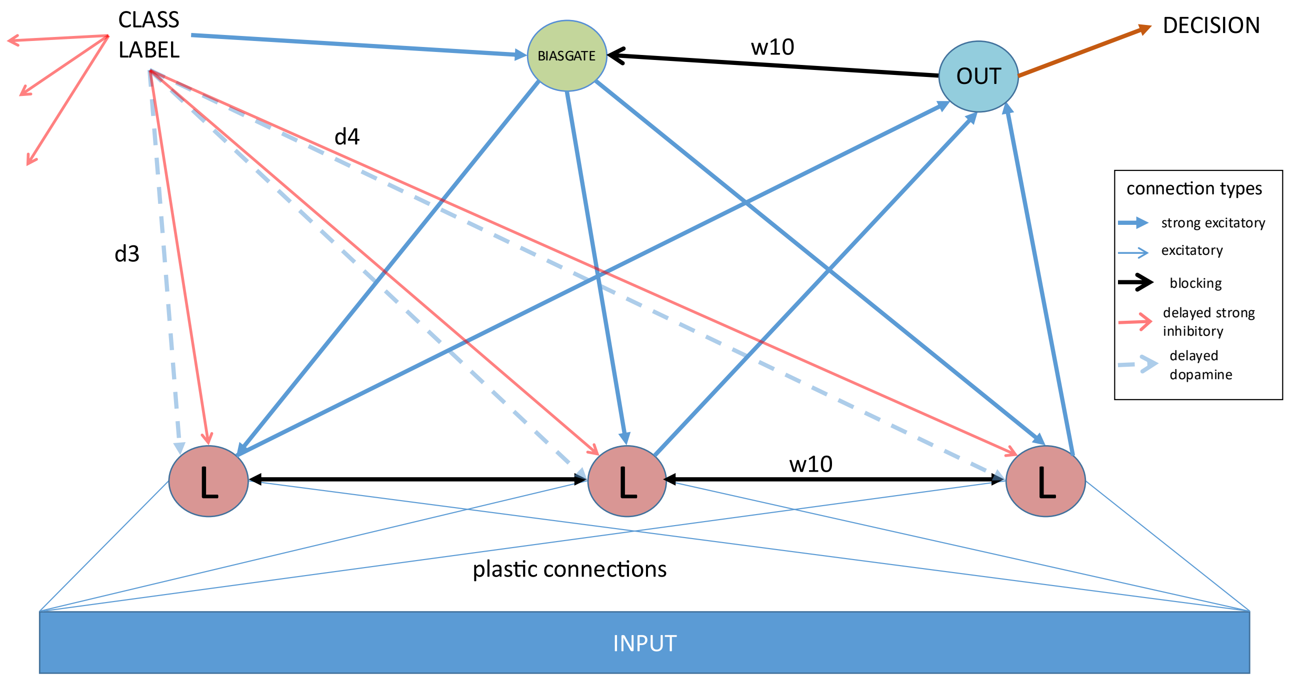
The images in the uncompressed form are stored in the binary file `MNIST.bin`. Image pixels are stored row by row, 1 byte per pixel. Image labels are stored in the text file `MNIST.target` – one line per image.

This supervised classification learning problem will be solved by the network specially designed for classification tasks. Its architecture is called CoLaNET (Columnar-Layered NETwork). The network structure defined in the file `3.nnc` is briefly described below.

The network consists of several identical structures called columns. One column corresponds to one target class. Thus, if we apply this network to MNIST there will be 10 columns in the network. Every column contains 3 kinds of neurons organized in 3 *layers*.



The structure of one column is shown below.



We see that it includes several neurons labelled by the letter L – which means "learning neurons". While one column corresponds to one target class, one L neuron corresponds to significantly distinctive instances (sub-classes) of one class. All neurons are described by the simplest LIF (leaky integrate-and-fire) model with slight modifications described later.

We begin the description with the working regime assuming that all L neurons have correct values of synaptic weights (which are the result of the learning process considered later). In this regime, the network behavior is simple. The object description has the form of spike trains emitted by input nodes (the blue rectangle at the bottom). If the L neurons have right values of their synaptic weights, then only an L neuron belonging to the correct column will fire in response to this stimulation. It will cause firing of the OUT neuron which signals presentation of an object of the corresponding class.

Now discuss the learning process. In this process, the network obtains the information about the current object as well as the information about class (label) of this object. As it was said, the latter has the form of a single spike emitted by the respective input node encoding the current class label just after the object presentation.

Here, we encounter the first significant distinction from the classification process in the traditional neural networks. The traditional formal neurons have no internal dynamics – their output depends only on the current values of their inputs and does not depend on the previous input values. In contrast, spiking neurons are dynamical systems – their state depends on their history. Therefore, in the case when the consecutive examples presented to the network are independent (as they are in typical machine learning tasks), the presentations of two consecutive examples should be separated by a certain period of "silence" – absence of any spikes. It is necessary to exclude influence of the previous object on classification of the current object. In our case, every example is presented during 10 msec, and the silence period which is 5 msec. Moreover, in order to completely erase information about past image, the active input node send powerful inhibitory signal to all L neurons. This signal propagates 4 msec so that it reaches L neurons just before the next image presentation. The L neurons have the strict lower boundary of their membrane potential equal to 0 that guarantees their reset after obtaining a strong inhibitory spike.

At the beginning of the learning process, all weights of all plastic synapses (only neurons from the lowest layer have plastic synapses) are zero. Therefore, the stimulation from the input cannot make them fire. However, the L neurons have two other sources of stimulation – their internal stochastic stimulation (implemented as small random number added to membrane potential every simulation step) and one of the class label nodes. The latter sends a spike to the BIASGATE neuron of its column (the 2nd layer). This spike forces the BIASGATE neuron to fire and send, in its turn, spikes to strong excitatory synapses of all L neurons in its column. If the L neurons were isolated, they would fire in response to such a powerful stimulation. But all L neurons of the same column are interconnected by strong blocking connections – they form the so called WTA (= "winner takes all") ensemble. The purpose of the WTA ensemble is to guarantee that at most one neuron would fire at each network simulation step. It is realized due to a special inter-neuron arbitrage mechanism. This mechanism compares membrane potentials of the neurons ready to fire and selects the neuron with the greatest potential value. To make neurons slightly unequal, the aforementioned internal stochastic stimulation is introduced in the ArNI-X neuron model. Due to this random stimulation, there is a random winner in the WTA ensemble who fires and suppresses all the other neurons in the ensemble. Beside the BIASGATE neuron, the spikes from the label input node reaches also L neurons – but with 3 msec delay (after one of the L neurons has fired). These spikes come to special synapses of the L neurons called "dopamine". But only the spike coming to the winning neuron has effect. This spike triggers the dopamine plasticity process in this neuron. In accordance with the dopamine plasticity rule, all plastic synapses having obtained spikes a certain time before neuron firing are potentiated if the neuron receives a dopamine spike shortly after that firing. As a result of this process, one learning neuron in the column corresponding to the class presented slightly potentiates synapses connected to the recently active input nodes. These weights are still insufficient for firing solely from the input stimulation. However, the next time when similar image will be presented, this winning neuron will probably have greater positive value of the membrane potential at the moment of obtaining the excitatory spike from the BIASGATE. Therefore, it will have high chances to become a winner again, thus further potentiating the same set of synapses. Here, we should mention another important feature of the ArNI-X plasticity model – the constancy of the total synaptic weight of one neuron. Whenever some synapses are strengthened, all the other synapses are uniformly weakened. It means that the neuron – winner not only becomes more sensitive to the first presented image but also becomes less sensitive to significantly different images. Thus, if the second presented image from the same class will have little resemblance with the first one, then the first winner will receive negative stimulation and, therefore, it will have less chances to win this time. Due to this mechanism, different L neurons in the same column learn to react to different instances of a target class.

After some number of the plasticity acts described above, some L neurons acquire the ability to fire in response to input stimulation without help of BIASGATE neurons. In this case, the firing L neuron stimulates the OUT neuron of this column. It fires and blocks the BIASGATE neuron for all period of current image presentation because stimulation from BIASGATE is not needed anymore.

It remains to say that this scheme also has a protection against wrong L neuron firing. In fact, the described plasticity model consists of two components – anti-Hebbian plasticity and dopamine plasticity. Dopamine plasticity was briefly described above. The anti-Hebbian plasticity mechanism is also simple. Whenever a neuron fires, all its plastic synapses having received a spike shortly before this are depressed. It is just the contrary to the original Hebbian law stating that all synapses helping the neuron to fire are potentiated. But in our case, the anti-Hebbian rule is needed. Indeed, L neurons should react only to the correct images. The correct images are marked by the activity of the respective class label node which delivers dopamine reward to the L neuron. If an L neuron fired and did not receive the dopamine reward, it fired wrongly and, therefore, the synapses which forced it to fire should be suppressed. Thus, the complete picture is the following. When an L neuron fires (and this firing is not forced by a strong non-plastic synapse) all its synapses which contributed to this firing are depressed. They remain depressed if nothing more happens. But if, afterwards, this neuron receives a dopamine spike these synapses are potentiated. Hence, three possible scenarios are possible:

- **The neuron did not fire during input stimulation and was selected as a target for stimulation from BIASGATE.** Only dopamine plasticity should work – to potentiate the synapses receiving spikes. It gives it chances to fire correctly next time.
- **The neuron fired during input stimulation but it was wrong (no dopamine reward).** Only anti-Hebbian plasticity works - synapses receiving spikes are depressed. It lowers the neuron's chances to fire wrongly next time.
- **The neuron fired during input stimulation but it was right (dopamine reward followed).** Both plasticity mechanisms work – but they work in the opposite directions so that nothing changes. The neuron works correctly – we should not modify it.

Now, we consider how this network is described in ArNI-X language. This description includes the majority of implemented in ArNI-X network and neuron features.

In this tutorial, the input spikes are not pure noise but encode information contained in files, therefore, the `RECEPTORS` node should describe it. Now, the input `type` is `image`. It means that the input file contains monochrome images stored in the uncompressed form one after one, 1 byte per pixel, by rows. The file name is specified in the `source` node. The file format is described in the node `Special`. It contains the tags `width` and `height`, defining the image size; `offset` in the file the image data starts from; time step count per image (`ntact_per_image`); time step count per image presentation – without silence period (`image_presentation_time`); and spike frequency corresponding to the maximum pixel brightness (= 255) – here, it is 1 kHz.

However, in our case, it is not the only source of spikes. The network uses for learning also class label spikes. Their source is another input spike module called `StateClassifier`. It reads data from the file specified in the `target_file` node. It is a text file where each line contains label for the respective image (in the same order as in input image file). Besides that, it is necessary to specify the learning period length (`learning_time`). Since we use 60000 images for learning and each image corresponds to 15 emulation steps, the learning time equals to 900000.

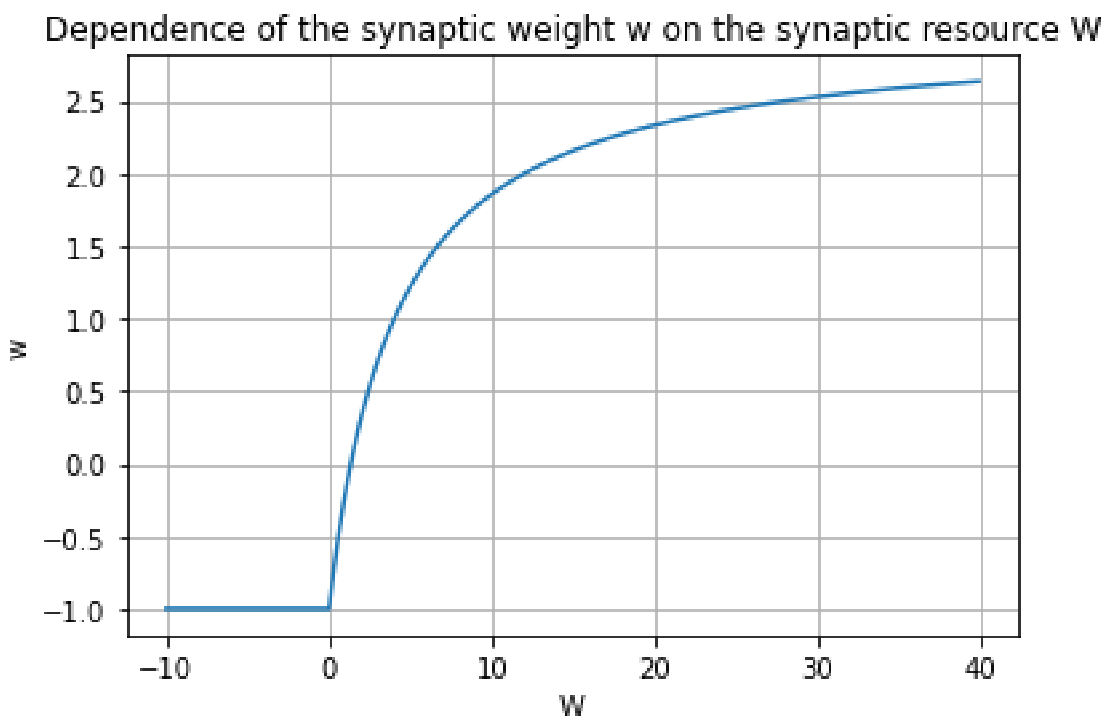
Now, let us consider the network itself. Strictly speaking, in this tutorial we use an ensemble of 15 networks with the similar structure instead of a single network. The decision of the whole ensemble is determined from votes of its members – every network votes and the majority wins. This measure helps eliminate rare accidental errors of individual networks. The ensemble size is set by the `ncopies` attribute of the `NETWORK` node.

The networks populations and projections are described in the subsequent `Section` and `Link` nodes. We see that description of the L population contains many new parameters. The majority of them determine synaptic plasticity of L neurons. In order to clarify their meaning, we describe the ArNI-X plasticity model in more detail.

The most important, in this 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. Different relationships between synaptic resource and synaptic weight are implemented in ArNI-X. In the given network, this dependence is expressed by 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 $-\infty$ to $+\infty$, 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. This feature is most useful for unsupervised learning. Later, we describe another mechanism fighting catastrophic forgetting, more applicable to supervised learning.

As it was said, there are several plasticity mechanisms in our model. The first one is Hebbian (or anti-Hebbian) plasticity. Donald Hebb's law of synaptic plasticity states that the synapses, which helped the neuron to fire, are potentiated (in the anti-Hebbian model, they are depressed). Since effect of a presynaptic spike on membrane potential decreases with time exponentially, we can conclude that synapses obtaining last spike long time (sufficiently greater than membrane potential decay time `chartime`) ago before neuron firing do not contribute to it. In our model, Hebbian plasticity affects the synapses obtaining spikes during last 3 * `chartime` msec before firing. But if a neuron emits spikes by dense packets (bursts or TSS - tight spike sequences) we treat every TSS as a single spike so that at most one weight modification of each synapse is caused by one TSS even if the TSS includes many spikes. All spikes separated by time ISI_{max} or less form one TSS. Our version of Hebbian plasticity is very simple. Synaptic resource of every synapse having obtained a spike $t_H = 3$ * `chartime` ago or less before TSS onset (or during the TSS) is increased by a constant not depending on exact moment of spike arrival. This constant d_H depends on the basic neuron plasticity value $\overline{d_H}$ 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.

Another component of ArNI-X synaptic plasticity mechanism is *reward or dopamine plasticity*. It is also very simple. When a neuron receives a spike via its reward synapse at most the time t_D after the last spike in TSS, the resources of all its plastic synapses having obtained a spike during this TSS or not earlier than the time t_H before the first spike in the TSS are changed by the value equal to the weight of this reward synapse. Again, every synaptic weight is changed at most once by one dopamine spike. The reward synapse weight may be positive or negative (as it is in our case). t_D is called *dopamine plasticity period*.

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 `weight_inc` 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`.

At last, we should mention two other features of the ArNI-X plasticity model.

Constant Total Synaptic Resource. In order to introduce competition between synapses inside one neuron, we added one more component to the model of synaptic plasticity – constancy of neuron's total synaptic resource. Whenever some synapses are depressed or potentiated due to the above mentioned plasticity rules all the other synapses are changed in the opposite direction by the constant value equal for all these synapses such that the total synaptic resource of the neuron is preserved. Effect of this rule can be controlled introducing imaginary unconnected synapses whose only role is to be a reservoir for the excessive (or additional) resource. The competitive effect is maximum when there are no such silent synapses and it vanishes with their number approaching infinity.

Lower Excitability of Trained Neurons. As it was said, the crucial CoLaNET property is multiple recognizers belonging to the same target class (column) which should recognize different instances of this class. To provide CoLaNET with this property, L neuron should be sufficiently selective i.e. sensitive to their sub-class of the target class and insensitive to the other sub-classes. The former requirement is fulfilled due to high positive weights of the synapses connected with the input nodes active during presentation of objects from the sub-class recognized. In order to satisfy the latter requirement, the trained neurons are made less sensitive in general due to elevated threshold potential. The indicator of a trained neuron is presence of synapses with high positive weight. Therefore, a logical choice is to make the threshold potential u_{THR} dependent on the sum of positive synaptic weights (of plastic synapses only):

$$u_{THR} = H + \alpha \sum \max(w_i, 0).$$

Here, α is a small positive constant, H – the constant component of the threshold potential.

But let us return to description of the L population. First of all, it is important that this population has a structure – it is not just a "flat" set of neurons but is subdivided into 10 columns as it was mentioned above. In our network, there are 20 neurons in each column. It can be said that L neurons form a 2-D lattice 10×20. This structure is similar to tensor organization of layers in traditional neural networks as they are described in the popular packages TensorFlow or Keras (but the role of this organization is completely different). To specify population structure, the node `Structure` is used. The structure type is set by the `type` attribute. Here, it has value "`L`" (lattice). The lattice dimensions are defined by the child `dim` nodes – from the lowest to the highest dimension.

In order to exclude influence of one image to the subsequent one, the class label spike sends powerful inhibition to all L neurons (red arrows on the CoLaNET scheme). However, to make the starting states of all neurons equal, it is necessary to limit their membrane potential from below. It is made by the `minpotential` node. Its value is 0, therefore, L neuron membrane potential cannot be negative.

The other parameters are relevant to the learning mechanism described above:

`stochastic_stimulation` – specifies strength of the neuron stochastic stimulation – every time step the membrane potential is incremented by a value randomly distributed in the range (0, `stochastic_stimulation`).

`weight_inc` – the Hebbian plasticity value. If it is negative, then plasticity is anti-Hebbian.

`dopamine_plasticity_time` - t_D from the plasticity rule description above. Here, it equals to 10 since the class label spikes comes on the 11th step after image presentation beginning.

`maxTSSISI` - ISI_{max} value (see above). Its value is 10 since there should be at most one Hebbian/dopamine plasticity act per synapse per image presentation.

`minweight` and `maxweight` - w_{min} and w_{max} , respectively.

`nsilentsynapses` – the number of the imaginary synapses consuming or supplying additional synaptic resource keeping its total value constant for one neuron.

`threshold_excess_weight_dependent` – the constant α in the formula for threshold membrane potential.

It was the description of the L population. The two other populations, OUT and BIASGATE, contain ordinary non-plastic LIF neurons – their descriptions are very short.

The sets of inter-neuron connections (projections) are described in the subsequent `Link` nodes. Non-plastic excitatory and inhibitory connections were considered in the previous tutorial, but in the present tutorial, three more connection types are used (specified by the `type` attribute of `Link`). These are `plastic` (the synapses which learn), `reward` (the dopamine synapses), and `gating` (the blocking or activating synapses).

For plastic synapses, the original distribution of their resource may be specified in the `IniResource` node. The distribution type may be specified by the attribute `type`. In our example, all synaptic resources are initialized by the value corresponding to zero synaptic weight.

The distinctive feature of CoLaNET is its structure. To support this structure, projections should obey certain requirements – they are called connections policies (set by the `policy` attribute). For example, mutual blocking connections between L neurons are organized in all-to-all manner but only within their column (between the neurons with the same indices inside lattice except the least significant one). This policy is `all-to-all-sections`. The class label input nodes are connected to L neurons by dopamine connections in such a way that the first node projects onto the first 20 L neurons, the second one – to the next 20 and so on. This is the policy `aligned`. But the strong inhibitory connections from class label to L obey the `all-to-all` policy.

It is not difficult to see that the projection descriptions correspond to the network structure described above.

The last thing specified in `3.nnc` is `Readout` – the module processing activity of the output neurons. Again, the dynamic library implementing this processing should be specified. Here, it is also `StateClassifier`. Its purpose – to evaluate accuracy of classification performed by SNN in the form of spikes from the `OUT` population. This population should be specified in the `output` node.

We launch the simulation by the command line

```
ArNICPU ..\Tutorials -e3 -f900000 -E900000:MNIST.nns.csv
```

The option `-f900000` tells that, at tact 900000, synaptic plasticity will be switched off. As we discussed above, it corresponds to the first 60000 images used for learning. The rest 10000 images will be used for testing the trained SNN, therefore, it should not use them for learning. The `-E` option is needed to export the network structure at tact 900000 (when it had learnt to classify MNIST digits) to the text file `MNIST.nns.csv`. We will use this comma-separated values (CSV) file to explore the plastic synapse weights.

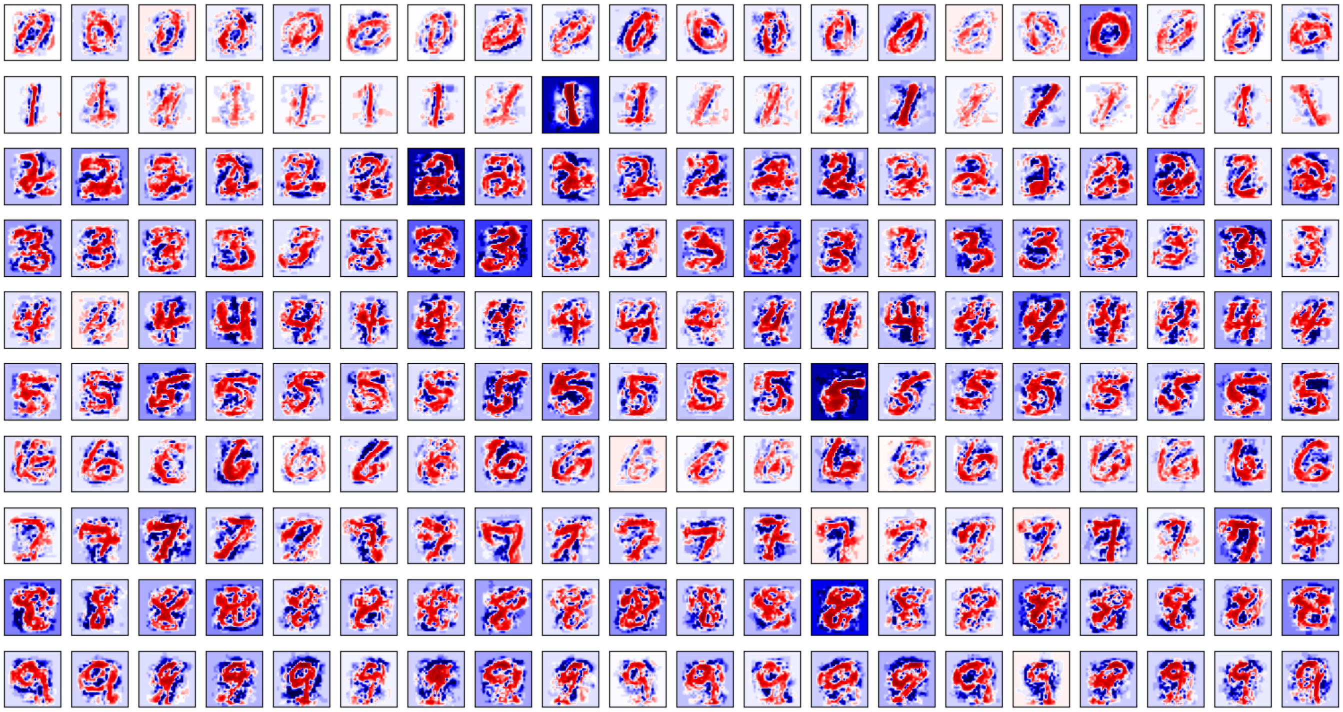
Since this network is much larger than the two previous ones, running this simulation on not very powerful central processor may take considerable time – 3 hours and more. For this reason, it is better to use GPU (launching `ArNIGPU` instead of `ArNICPU`), if it is available.

Upon completion, the simulator returns the termination code. If `StateClassifier` is used as a readout, the termination code is determined by `StateClassifier`. It is classification accuracy (measured on the test subset) multiplied by 10000. We see that our network classifies MNIST digits with the accuracy 96.07%. It seems to be not very high result comparatively to modern convolutional networks. However:

1. Only one training epoch is used in this tutorial (it is because SNNs are often used in incremental learning scenarios, where repetitive presentations of the same examples is usually impossible).
2. The network is quite simple – it includes only 3300 neurons (the MNIST winners are much larger).
3. It does not include convolutional layers – use of convolutional SNN would make this tutorial more difficult for comprehension.

Therefore, in fact, 96.07% accuracy is a good result.

It is interesting to look at the distribution of plastic synapse weights of L neurons recognizing different digits. Their values are contained in `MNIST.nns.csv` and can be extracted from there by the Python script `MNISTweights.py`. Since out network is an ensemble of 15 similar networks, we will look at only one of them depicting the weights of its 200 L neurons. Each row of the figure corresponds to one digit.



Red color encodes positive weights, blue – negative. We see that different L neurons recognize different writings of the respective digits – as it was expected for the CoLaNET architecture.

NEURON MODEL

The neuron model implemented in this package is a generalized version of 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 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. 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

$$\begin{cases} \dfrac{du}{dt} = -\dfrac{u}{\tau_v} + \sum_{i,j} w_i \delta(t - t_{ij}) + I_{noise} \\ \dfrac{du_{THR}}{dt} = -a\,\mathrm{sgn}(u_{THR} - H) + \sum_k \hat{T}\delta(t - \hat{t}_k) \end{cases}$$

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 (if the neuron is currently active – see below) and the current value of u_{THR} is subtracted from u. All potentials are rescaled so that after the long absence of presynaptic spikes $u \rightarrow 0$ and $u_{THR} \rightarrow H = 8.531$ (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. The neuron may have source of its internal stochastic stimulation I_{noise}. It is implemented as a random number uniformly distributed in the range [0, s_{noise}] which is added to u every simulation step.

Out implementation of LIFAT has two additional features. 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 threshold potential is too high. It is equivalent to 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 higher and higher. 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 A 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. Besides the gating synapses, a neuron can be inactivated by its own firing if its refractory period τ_R is positive. In this case, each neuron firing makes it's A equal to -τ_R.

SYNAPTIC PLASTICITY RULES

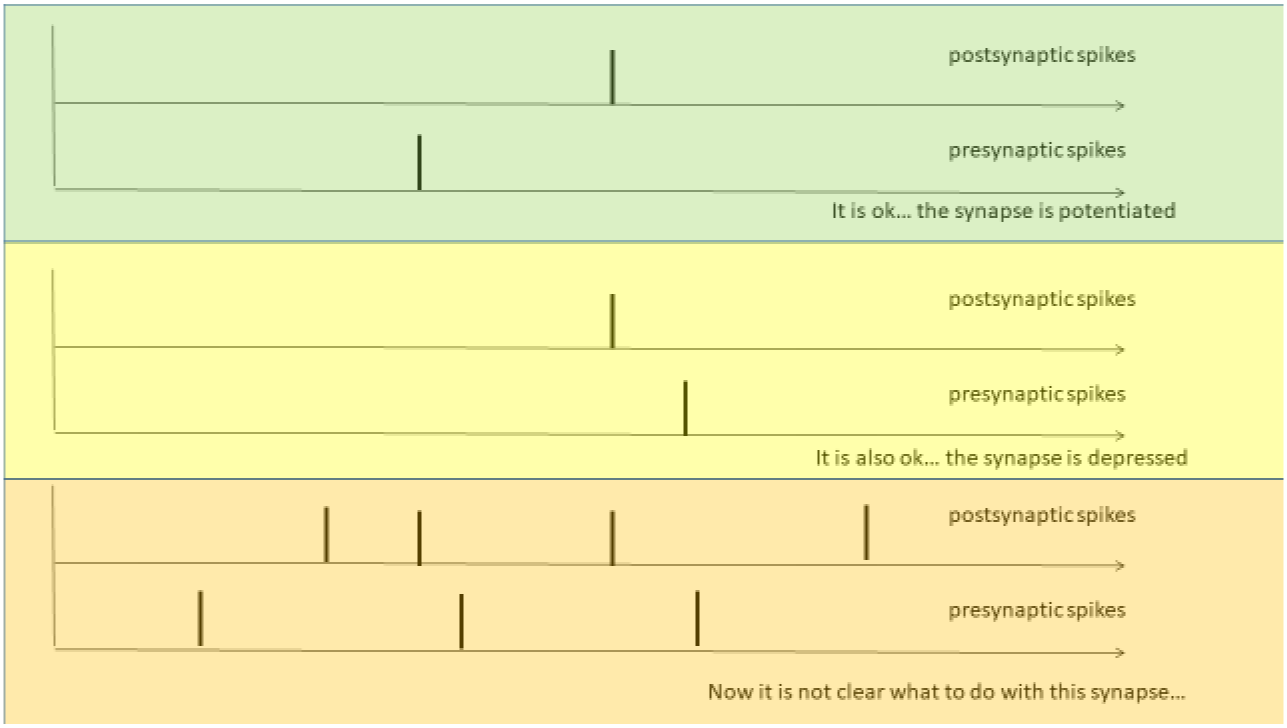
The most valuable ability of neural networks is their ability to learn. Learning in the traditional non-spiking neural networks (artificial neural networks – ANN) is implemented due to the appropriate modification of synaptic weights of their neurons. In this sense, the SNNs do not differ from the ANNs – 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 only.

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 or prevent firing. If a neuron fires when it receives a spike via its fixed synapse, we will call it *forced* firing.

The ArNI-X local synaptic plasticity rules (there are two kinds of 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 other requires 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. Usually, 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.

Besides that, there may be anti-Hebbian variant of this plasticity mechanism – depressing the involved synapses instead of potentiating them.

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 changed 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 (or punish) 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 modified. Effect of one spike to membrane potential decays with the time constant τ_v, hence $T_H = \beta\tau_v$., where β is a small number (3 – by default).
3. All synaptic resources are changed by the same value d_H independently of exact timing of presynaptic spikes. d_H may be negative as well as positive.

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. However, in contrast with Hebbian plasticity, implementation of dopamine plasticity is asymmetric – reward and punishment use different mechanisms called 2-factor and 3-factor plasticity rules. Choice of 2- or 3-factor rule is determined by the sign of the total weight of all dopamine synapses obtaining spikes at the given moment.

2-factor Plasticity (Punishment)

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 dopamine 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.

3-Factor Plasticity (Reward)

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. 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 (see above).

This asymmetry is a consequence of the basically different semantics of punishment and reward. If some neuron group should react by their activity to certain conditions, it means that each neuron in the group should recognized its own subset of these conditions (otherwise, several neurons would be functionally equivalent and, therefore, could be replaced by a single neuron). That is, if neuron firing is prohibited under certain conditions expressed by combination of synapses obtaining a spike that is indicated by a punishment spike, then the respective synapses of all neurons in the groups should be depressed (2-factor rule). But if some neuron recognized the target conditions correctly by firing then only this neuron should be rewarded (3-factor rule).

It is necessary to say that Hebbian plasticity and dopamine plasticity can be combined inside one neuron.

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 in the following way. 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 additional 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 $D\min\left(2^{-s}, 1\right)$, 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 most recent firing was forced	The R sign	The stability change
No	Negative	rR
Yes	Negative	0
No	Positive	$2rR$
Yes	Positive	$-rR$

These rules may seem 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 – $2rR$. (in case of balanced anti-Hebbian and positive dopamine plasticity - $R = -\overline{d_H}$)

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

Lower Excitability of Trained Neurons

This feature was considered in Tutorial 3. The neuron's threshold potential can be made dependent on the sum of positive synaptic weights (of plastic synapses only):

$$u_{THR} = H + \alpha \sum \max(w_i, 0),$$

where, α is a small positive constant. In this case, its own dynamics are eliminated $\hat{T} = a = 0$.

Synaptic Weight Quantization

In Tutorial 3, we discussed the relationship between the synaptic weight *w* and the synaptic resource *W*. However, the formula given there, expressing a smooth dependence of *w* on *W*, is not the only variant of this relationship implemented in ArNI-X. The other variants are needed mostly in the case when the SNN are destined to be placed on a specialized neurochips where individual microprocessors (neurocores) have very limited number computational abilities. In this case, either this dependence should be much simpler or set of possible weight values should be very limited (weight quantization). To support this, three other dependences of *w* on *W* are implemented in ArNI-X in addition to the above-mentioned one (called `smooth`):

- `clipped` – $w = \max(w_{min}, \min(w_{max}, W))$;
- `quantized` – $w = \begin{cases} w_{min}\ if\ W < 0 \\ Q_{\lfloor log_2(W) \rfloor}\ if\ log_2(W) < |Q| + 1 \\ w_{max}\ o/w \end{cases}$, where Q is the set of possible weight values – these values are selected from the condition of minimality of the maximum difference between the weights calculated in this model and in the model `smooth`;
- `superquantized` – $w = \begin{cases} w_{min}\ if\ W < 0 \\ \frac{w_{min}}{2} if\ W < 2^q \\ 0\ if\ 2^q \le W < 2^{q+1} \\ \frac{w_{max}}{2} if\ 2^{q+1} \le W < 2^{q+2} \\ w_{max}\ o/w \end{cases}$, where q is a small positive integer determining quantization step.

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 simulated 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`. This node may have one attribute `model` determining relationship between synaptic resource and synaptic weight (see the previous section).

Inside the `SNN` node, there should be one or more `RECEPTORS` nodes, `NETWORK` nodes, and, optionally `Readout` node, which describe the input nodes (there may be several input node sections sending signals with different semantics), the components of the network, and the module obtaining output spikes, respectively.

`RECEPTORS` – Description of Input Nodes

A `RECEPTORS` node should have the attribute `name` defining name of the input node section, and may have the attribute `n` specifying number of input nodes in this section. If the latter is not set, it should be set by the module implementing this input node section (see below).

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 two such libraries – `fromFile`, which can read input from a file and add Poissonian noise, and `StateClassifier`. The latter is used in classification tasks - it provides spikes labelling current target class(es) and measures the network classification accuracy.

An `Implementation` node should include only one node with the tag `args` with the attribute `type`.

For `fromFile`, this attribute can take the following values:

- `none` – No external input data (only Poissonian noise).
- `text` – The input spikes are read from a text file. One line corresponds to one simulation step. 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 a 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 simulation step (one input signal portion) should have 64 bit (8 byte) alignment. Bit records for successive simulation steps 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` sub-node in the case of the `fromFile` module 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 parameter 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.
- `shuffle` – if present, this node instructs the system to shuffle the input file contents (text lines, binary data portions or images) randomly. Optional random number generator seed value may be specified in it.
- `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).
    - `image_presentation_time` - time of presentation of one image (msec).
    - `maxfrequency` (optional) – the maximum input node firing frequency (kHz). Since brightest pixels have value 255, the constant *g* equals to this value divided by 255. The default value is 1.

    - ntact_per_image – time corresponding to one image presentation (msec). If this value is greater than image_presentation_time, then during the rest simulation time, the input nodes do not emit spikes.
    - 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 (simulation steps) recorded in the file which should be skipped (from the file beginning).

If the input node section is implemented by the StateClassifier module, the args sub-node can contain the following parameters:

- criterion – the classification accuracy evaluation. It determines how the simulation return code is evaluated (usually it equals to number of simulation steps executed, but in the case of StateClassifier readout, its meaning is different). The possible values are:
    - absolute_error – returned code is the fraction of correctly classified test examples multiplied by 10000.
    - weighted_error – returned code is the mean fraction of correctly classified test examples in each target class multiplied by 10000.
    - averaged_F – returned code is the sum of the F measures of classification accuracy in all classes weighted by example count in each class and multiplied by 10000.
- learning_time – this simulation period from the simulation beginning is not used for the accuracy evaluation.
- max_silence – the maximum allowed time period without spikes from the output populations. If the silence is longer, the simulation is terminated.
- no_class – there may be several such nodes. They specify the target classes which will be ignored – the examples labelled by these strings will be considered as not belonging to any class.
- prediction_file – the detail information on classification of test examples will be stored in this file.
- sequential_test – if this node is present, it means that objects from one class are often presented in series. Therefore, in case of unclear prediction, it is reasonable to assign to the current example the same label as for the previous example.
- shuffle – it has the same meaning as for fromFile and should correspond it.
- spike_period – the first class label spike will be emitted after this time since the example presentation offset and after that will be emitted with this period. The default value is 10 that corresponds to the object presentation time in the CoLaNET protocol (see Tutorial 3).
- state_duration – the total time period corresponding to one example presentation. The default value is 15 that corresponds to the CoLaNET protocol (see Tutorial 3).
- states_to_skip – the number of lines in the beginning of the target class file which should be skipped (from the file beginning).
- target_file – the file containing example class labels – one line per example.

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, there may be several NETWORK nodes and they can include other kinds of sub-nodes. The latter case corresponds to SNN sections implemented by dynamic libraries using the respective API, which is not covered by this manual, and, therefore, 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).

Also, there is a special case of NETWORK node without sub-nodes and with the attribute saved. Value of this attribute is a name of a file containing a previously saved SNN (with the extension .nns – see below). In this case, the network will be restored from the file.

The NETWORK node may have the attribute ncopies. If it exists, several instances of the network described by the NETWORK node will be created. Their count is the integer value of the attribute ncopies. The postfix #<instance number> will be added to population names in these instances in order to make the population names unique.

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 (see the notation used in the neuron model description):

- 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).
- hebbian_plasticity_chartime_ratio – β (default = 3).
- 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.
- `nwquants` – the number of intermediate synaptic weight values (not including w_{min} and w_{max}) in the `quantized` synaptic weight quantization model.
- `refractory_period` – τ_R.
- `stochastic_stimulation` – s_{noise}.
- `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 – *tensor (lattice) structure*, corresponding to the value of `type` equal to `L`. For this structure type, the `Structure` node includes one or several nodes `dim` whose values should be integer numbers. These numbers define dimensions of the lattice whose nodes correspond to neurons from this population. Therefore, the product of all these numbers should be equal to the population size.. This structure is taken into account when inter-neuron connections are created (see below).
- `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_excess_weight_dependent` – α.
- `threshold_inc` – the threshold potential increment $\hat{T}$.
- `weight_inc` – the basic level of Hebbian plasticity $\overline{d_H}$.

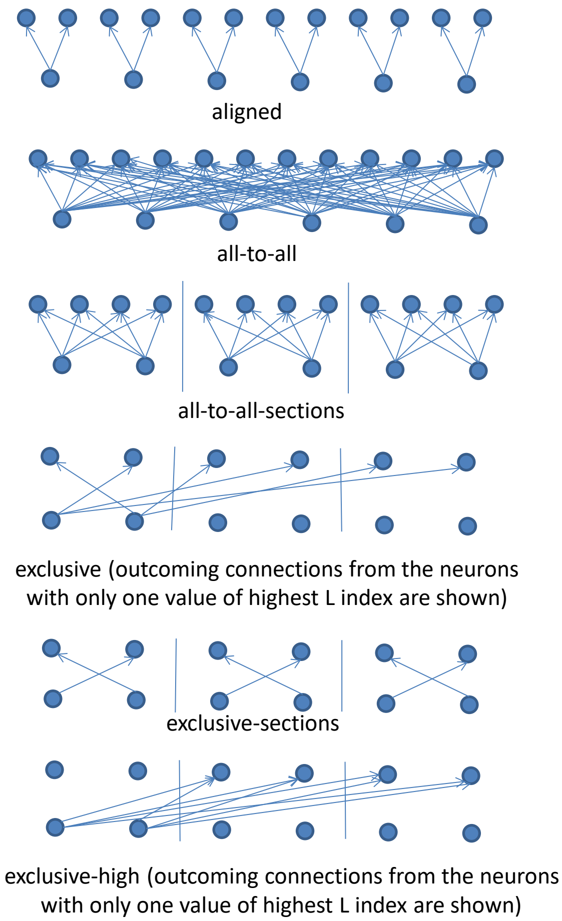
`Link` – Projection Property Definition

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 `L` structure (see the tag `Structure` inside `Section`) and the dimensions of these structures except the lowest one are the same. In this case, all-to-all connections among the neurons with the same sets of the `L` structure indices except the lowest one are created. For example, winner-takes-all blocking lateral connections can be made using this policy.
    - `exclusive`. If both connected populations have `L` structure with the same lowest dimension, then these are all-to-all connections excluding connections between the neurons in with the same lowest `L` structure index. Otherwise, these are all-to-all connections excluding connections between the neurons with the same index inside their populations.
    - `exclusive-high`. This connection policy is only allowed if both connected populations have `L` structure with the same highest dimension. These are connections between the neurons with the different highest `L` indices.
    - `exclusive-sections`. This connection policy is only allowed if both connected populations have the same `L` structure. These are connections between the neurons with the different highest `L` indices and the same values of all the rest indices.

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

These connection policies are illustrated on the picture. Four lower schemes illustrate connections between populations with `L` structure. Different sections correspond to different higher L dimension indices while different positions inside sections – to lower dimension indices.



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).

SIMULATOR COMMAND LINE ARGUMENTS

The emulator command line has the syntax `ArNI(C|G)PU` <ExperimentSeriesDirectory> <Options>*. `ArNICPU` preforms the simulation 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<IterationStep>:<FileName> - used to export network configuration at IterationStep to FileName. If this option is used, the current directory should contain the dynamic library NetworkExporter, performing the network state export. The default version of this library shipped with this distribution package saves the network state in CSV format.

-`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.