kaspersky

Kaspersky Neuromorphic Platform

© 2025 AO Kaspersky Lab

Contents

About Kaspersky Neuromorphic Platform
- Distribution kit
- Hardware and software requirements
- Third-party libraries

Kaspersky Neuromorphic Platform architecture
- Neural network description format
  - The structure of the file describing neuron groups
  - The structure of the file describing synapse groups
- Data exchange mechanism in Kaspersky Neuromorphic Platform
- Data formats processed by the platform
- About Kaspersky Neuromorphic Platform backend
  - Supported backends
  - About the AltAI-1 neural processing unit
  - About CPU backend operation

Installing and uninstalling the platform
- Installing deb packages
- Installing Python development packages
- Scenario for building a platform project or an application solution
  - Downloading and unpacking the platform source code archive
  - Defining the Boost_ROOT setting in Windows
  - Configuring the project build settings for the application solution in C++
  - Configuring a build of a platform or an application solution in Windows
- Uninstalling the platform

C++ platform components
- neuron_traits library
- synapse_traits library
- core library
  - UID class
  - uid_hash function
  - continuously_uid_generator class
  - TagMap class
  - BaseData structure
  - Population class
  - Projection class
  - Backend class
  - Device class
  - MessageBus class
  - MessageEndpoint class
  - Subscription class
  - tags namespace
  - messaging namespace
- meta library
- devices library
- C++ framework
  - sonata namespace
  - coordinates namespace

synchronization namespace
storage namespace
population namespace
projection namespace
input namespace
output namespace
monitoring namespace
Network class
Model class
BackendLoader class
ModelLoader class
ModelExecutor class
Solving common tasks in C++
Adding a new neuron type
Adding a new synapse type
Execution of a neural network created from projections and populations
Python platform components
neuron_traits module
synapse_traits module
core module
UID class
uid_hash class
UUID class
continuously_uid_generator class
TagMap class
BaseData structure
Set of Population classes
Set of Projection classes
Backend class
BackendWrapper structure
MessageBus class
MessageEndpoint class
Set of Subscription classes
messaging namespace
Python framework
ANN2SNN package
Ternary layers of the ANN2SNN package
sonata namespace
input namespace
output namespace
BackendLoader class
Network class
Model class
SpikeMessageObserver class
ImpactMessageObserver Class
ModelLoader class
ModelExecutor class
Solving common tasks in Python

Execution of a neural network manually uploaded to the backend
A script for executing a neural network from ternary layers for image classification
Creating and training a neural network from ternary layers for image classification
Running a neural network inference from ternary layers for image classification
A script for executing a regression model with training blocks that encode and decode data
Creating and training a regression model
Running the regression model inference
Using the Kaspersky Neuromorphic Platform API
Participation in the development of the platform
Platform licensing
About data provision
Appendix. Settings and an example YAML file containing the configuration of the AltAI-1 neuromorphic processing unit
Glossary
AltAI ANN2SNN backend
AltAI SNN backend
Artificial Neural Network (ANN)
Backend
Computer
Connection
Cycle
Dense sequence of spikes (DSS)
DSS period
External environment
Framework
Modifier
Neuron
Neuron function
Population
Projection
Spike
Spiking Neural Network (SNN)
Synapse
Synapse function
Synapse weight
Synaptic impact
Tick
Information about third-party code
Trademark notices

About Kaspersky Neuromorphic Platform

The Kaspersky Neuromorphic Platform ("KNP" or "platform") is a software platform for developing, training and executing spiking neural networks on a variety of computers.

You can use Kaspersky Neuromorphic Platform to do the following:

- Create spiking neural networks (Spiking Neural Network, SNN) and train these on various types of input data, such as telemetry, events, images, 3D data, sound, and tactile data.
- Convert artificial neural networks (ANNs) into spiking neural networks and train them.
- Optimize the structure of the loaded neural networks.
- Conduct applied research in the field of input data classification and other application fields of neural networks.
- Develop new neural network topologies, for example, impulse analogs of convolutional neural networks that involve convolution in space and time.
- Develop new models of synaptic plasticity.
- Implement new neuron models.
- Implement application solutions based on neuromorphic spiking neural networks in the field of robotic manipulators, Internet of Things, unmanned systems, human-machine interaction, wearable devices, and optimization planning.
- Implement application solutions on devices with low power consumption or using neuromorphic processors.

To solve the above tasks, you can use the C++ and Python® languages. The platform supports CPUs as well as the AltAI-1 neuromorphic processing unit designed for power-efficient execution of neural networks on various classes of smart devices.

Distribution kit

The Kaspersky Neuromorphic Platform has several distribution kit options. You can choose the option that best suits your objectives for the platform.

The Docker image `knp-build-image` is also provided for building the platform or an application. The Docker image must be downloaded from the Docker Hub repository.

Kaspersky Neuromorphic Platform distribution kit for the development of application solutions in Linux

The distribution kit includes the following files:

- `pkg-deb.zip` archive with the following packages:
  - `knp-cpu-single-threaded-backend_<knp_version>_amd64.deb`: a deb package with CPU single-threaded backend binaries

- `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`: a deb package with CPU multi-threaded backend binaries
- `knp-cpp-framework_<knp_version>_amd64.deb`: a deb package with C++ framework binaries, which contains the following:
  - Binary code of the platform libraries
  - `LICENSE.txt` file containing the platform terms of use in English
  - `README.md` file containing the information about the platform in English
- dev-deb packages for each deb package with C++ binary code:
  - `knp-cpu-single-threaded-backend-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`
  - `knp-cpu-multi-threaded-backend-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`
  - `knp-cpp-framework-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-cpp-framework_<knp_version>_amd64.deb`

    Each dev-deb package includes the following files:
    - Header files
    - CMake modules for building the platform in C++
    - Static versions of the platform libraries
- `knp-python3-framework_<knp_version>_amd64.deb`: a deb package containing a Python framework for the `AltAI SNN` backend
- `knp-examples_<knp_version>_amd64.deb`: a deb package with Kaspersky Neuromorphic Platform usage examples
- `knp-documentation_<knp_version>_all.deb`: a deb package with the API reference guide
- `knp-unspecified_<knp_version>_amd64.deb`: a deb package that includes CMake service

- `knp-altai-backend_<motiv_version>_amd64.deb`: a deb package that contains AltAI SNN backend binaries

  The deb package is provided by Motiv NT.
- `knp-altai-backend-dev_<motiv_version>_amd64.deb`: a dev-deb package for `knp-altai-backend_<motiv_version>_amd64.deb`

  The dev-deb package is provided by Motiv NT.
- `knp_ann2snn.deb`: a deb package with the `ANN2SNN` package, AltAI ANN2SNN backend binary code for the AltAI-1 neuromorphic processor, and a component for placing the neural network on AltAI-1

  The deb package is provided by Motiv NT.

  where:
  - `<knp_version>` is the Kaspersky Neuromorphic Platform version

- `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package

Kaspersky Neuromorphic Platform distribution kit for the development of application solutions in Python for Linux

The Kaspersky Neuromorphic Platform distribution kit includes the following files:

- `pkg-deb.zip` archive with a `knp-python3-framework_<knp_version>_amd64.deb` deb package that contains a Python framework for the `AltAI SNN` backend
- `pkg-python.zip` archive with a `knp-<knp_version>-py3-none-any.whl` whl package that contains a Python framework for the `AltAI SNN` backend
- `knp_ann2snn.deb`: a deb package with an `ANN2SNN` package, `AltAI ANN2SNN` backend binaries for the AltAI-1 neuromorphic processor, and a component for placing the neural network on AltAI-1

  The deb package is provided by Motiv NT.
- `knp_ann2snn-<motiv_version>-py3-none-any.whl`: a whl package with an `ANN2SNN` package, `AltAI ANN2SNN` backend binaries for the AltAI-1 neuromorphic processor, and a component for placing the neural network on AltAI-1

  This package can be installed on an x86-based computer.

  The whl package is provided by Motiv NT.
- `knp_ann2snn_aarch64-<motiv_version>-py3-none-any.whl`: a whl package with an `ANN2SNN` package and the `AltAI ANN2SNN` backend binaries for the AltAI-1 neuromorphic processor

  This package can be installed on an ARM-based computer.

  The whl package is provided by Motiv NT.

  where:

  - `<knp_version>` is the Kaspersky Neuromorphic Platform version
  - `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package

Kaspersky Neuromorphic Platform distribution kit for working with platform source code

The distribution kit can be used for developing application solutions in C++ on Linux and Windows.

You can obtain Kaspersky Neuromorphic Platform source code files by cloning the platform repository. You can also obtain the source code files as an archive.

The distribution kit includes the following files:

- Platform source code
- API reference guide
- `README.md` file containing the information about the platform in English
- `LICENSE.txt` file containing the platform terms of use in English

- `NOTICE.txt` file containing the third-party code information in English
- `CONTRIBUTING.md` file containing the repository participant rules in English
- `CLA.md` file containing the repository participation terms in English

Hardware and software requirements

For Kaspersky Neuromorphic Platform operation, the computer must meet the following minimum requirements.

Minimum hardware requirements:

- CPU: Intel Core i5 or higher compatible processor
- Neuromorphic processing unit (if needed): AltAI-1
- 8 GB of RAM
- Available hard drive space:
  - 1 GB for installing Kaspersky Neuromorphic Platform.
  - 10 GB for building the platform or an application solution build.

Supported operating systems:

- Debian GNU/Linux 12.5 or later
- Ubuntu 22.04 LTS or later
- Windows 7
- Windows 10

You can use the device running any other operating system from the Linux family, if the operating system distribution kit contains the Boost library version 1.81 or later.

To work with the platform, the following software must be installed on the device:

- Visual Studio 2022

  This software must be installed for building the platform or application solution in Windows.
- Boost 1.81 or later

  When working in Windows, it is recommended to install the library precompiled for Visual Studio 2022 (compiler version: 14.3).
- CMake 3.25 or later
- Before installing platform component deb packages for an AltAI ANN2SNN backend, make sure that the device has the following software:
  - TensorFlow 2.4.1 or later

- Keras 2.3.1 or later
- NumPy 2.4.1 or later

  The NumPy library version must match the TensorFlow library version.

- Before installing a whl or deb package containing a Python framework for an AltAI SNN backend, make sure that the device has Python 3.10 or later.
- Before installing a whl package containing a Python framework for an `AltAI ANN2SNN` backend, make sure that the device has the following software:
  - NumPy 1.24.3
  - TensorFlow 2.13.1
  - Loguru 0.7.2
  - PyYAML 6.0.1
  - NetworkX 3.1
  - Matplotlib 3.6.3
  - tqdm 4.66.5
- Before installing a deb package containing a Python framework for an `AltAI ANN2SNN` backend, make sure that the device has the following software:
  - python3-pip
  - python3-numpy
  - python3-yaml
  - python3-networkx
  - python3-matplotlib
  - python3-loguru
  - python3-tqdm
  - TensorFlow 2.13.1

Third-party libraries

Kaspersky Neuromorphic Platform uses the following libraries:

`Boost`

Libraries are used to implement classes and structures (for example, `UID`, `BackendLoader`).

`FlatBuffers`

The library is used for serialization in Kaspersky Neuromorphic Platform.

GoogleTest

The library is used for platform tests.

Intel PCM

The library is used to receive device metadata (for example, power consumption data) from the CPU.

spdlog

The library is used for logging the Kaspersky Neuromorphic Platform operation.

Kaspersky Neuromorphic Platform architecture

Kaspersky Neuromorphic Platform includes the following components:

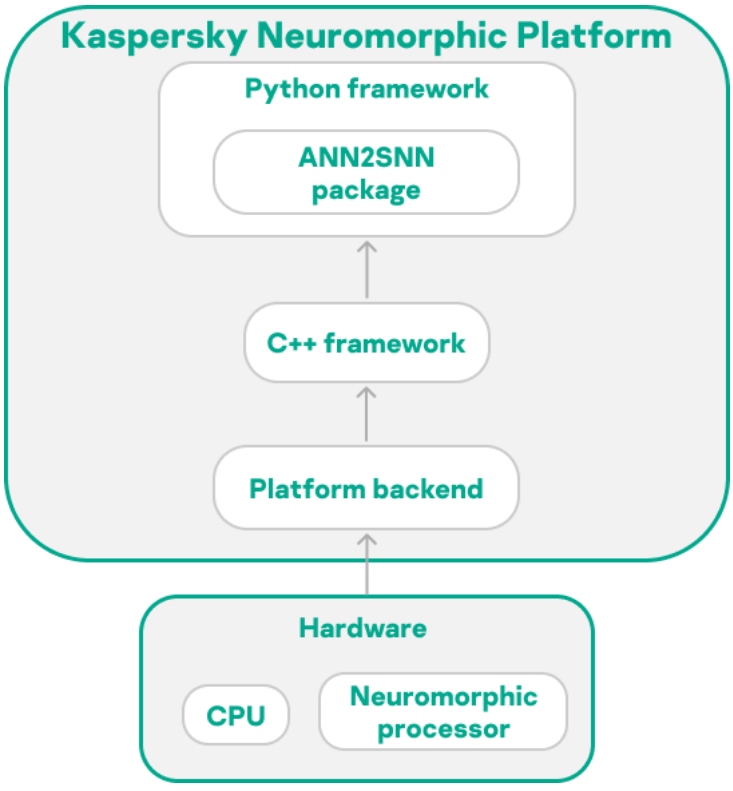
- *Python framework*. It is designed for creating spiking neural networks and converting artificial neural networks into spiking networks. Resulting spiking neural networks are trained with synaptic plasticity methods and can be executed on a central processor, an AltAI-1 neuromorphic processing unit, or a software emulator of the latter. Artificial neural networks converted to spiking networks are trained through backpropagation that considers the limitations of the AltAI-1 neuromorphic processing unit. Resulting neural networks are adapted for execution on the AltAI-1 neuromorphic processing unit, or a software emulator of the latter.

  The Python framework includes modules whose functionality overlaps with that of the C++ framework and an `ANN2SNN` package required for converting artificial neural networks into spiking neural networks and executing these on an AltAI ANN2SNN backend.

- *C++ framework*. It is designed for creating spiking neural networks and training these with synaptic plasticity methods. Resulting neural networks can be executed on a central processor, an AltAI-1 neuromorphic processing unit, or a software emulator of the latter.

- *Backend*. Starts training and execution of a spiking neural network on various computing devices (for example, neuromorphic processor AltAI-1). The platform provides several independent backends, each letting you train and execute spiking neural networks on an appropriate device type.

The figure below shows a diagram of the interaction between the Kaspersky Neuromorphic Platform components.



Kaspersky Neuromorphic Platform architecture

Neural network description format

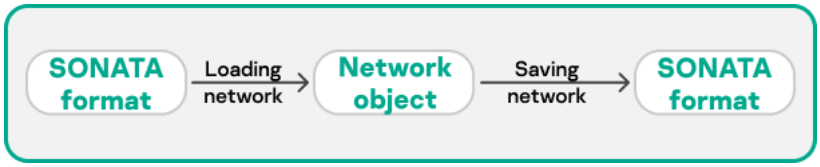
Kaspersky Neuromorphic Platform uses the `SONATA` format to describe the topology of the neural network.

In the `SONATA` format, the neural network is represented as a graph consisting of vertices corresponding to neurons and edges connecting the graph vertices and corresponding to synapses. The types of the corresponding populations and projections are assigned to neurons and synapses. The settings assigned to neuron or synapse types are automatically assigned to all neurons or synapses of these types. In addition, the attributes that define various neural network aspects, such as position, type, and model parameters, can be assigned to neurons, synapses, and their types.

A neuron population consists of one or several groups of neurons with a uniform tabular structure. The populations are serialized into HDF5 files and linked together using a CSV file describing the neuron types and the attributes applicable to these types. The groups of neurons are represented as the groups of HDF5 files containing data sets. The lengths of these data sets correspond to the number of neurons in the groups. Data in HDF5 files is stored in binary form.

Similar to neurons, synapses are defined by projections stored in HDF5 files. These files describe the attributes of each synapse. A synapse projection consists of one or several groups of synapses with a uniform tabular structure. Each HDF5 file is associated with a CSV file that defines the synapse types and attributes that apply to these types of synapses.

The figure below shows a diagram for loading a neural network object from the SONATA format and saving the network topology in the SONATA format.



Loading and saving a neural network

The structure of the file describing neuron groups

Below is an example of the HDF5 file structure describing neuron groups:

```
<file_name>.h5
└ nodes
  └ <population_name>
  ├ node_type_id
  ├ node_id
  ├ node_group_id
  ├ node_group_index
  └ <neuron_group>
  ├ <group_attribute>
  ├ ...
  └ dynamics_params
  ├ <dynamic_attribute>
  └ ...
```

The table below shows the data items in the file describing the neuron groups.

The data items in the file describing neuron groups

Data item	Description
`nodes`	Mandatory item for defining groups.
`< population_name >`	Mandatory item containing the name of the population. There can be several populations.
`node_type_id`	Mandatory item containing a data set with an array of IDs of the neuron types in the population.
`node_id`	Mandatory item containing a data set with the indices of the neurons in the population.
`node_group_id`	Mandatory item containing a data set with the indices of the groups to which the neurons in the population belong.

node_group_index	Mandatory item containing a data set with the indexes of the neurons in the corresponding groups.
< neuron_group >	Mandatory item containing the ID of the neuron group.
< group_attribute >	One or several attributes of the group. The attributes are optional and depend on the neuron type. Each attribute is a data set.
dynamics_params	A group of optional dynamic attributes.
< dynamic_attribute >	One or several dynamic attributes. Each attribute is a data set with attribute values for each neuron in the group.

The structure of the file describing synapse groups

Below is an example of the HDF5 file structure describing synapse groups:

```
<file_name>.h5
└ edges
  └ <projection_name>
  ├ source_node_id
  ├ edge_group_id
  ├ edge_group_index
  ├ target_node_id
  ├ edge_type_id
  ├ indices
  │ ├ source_to_target
  │ │ ├ node_id_to_range
  │ │ └ range_to_edge_id
  │ └ target_to_source
  │ ├ node_id_to_range
  │ └ range_to_edge_id
  └<synapse_group>
  ├ delay
  ├ syn_weight
  └ dynamics_params
```

The table below shows the data items in the file describing the synapse groups.

The data items in the file describing synapse groups

Data item	Description
edges	Mandatory item for defining groups.
< projection_name >	Mandatory item containing the name of the projection. There can be several projections.
source_node_id	Mandatory item containing a data set with the identifiers of presynaptic neurons for each synapse in the projection.
edge_group_id	Mandatory item containing a data set with the identifiers of the groups to which the synapses in the projection belong.
edge_group_index	Mandatory item containing a data set with the indices of the synapses in the corresponding groups.
target_node_id	Mandatory item containing a data set with the identifiers of postsynaptic neurons for each synapse in the projection.
edge_type_id	Mandatory item containing a data set with the type identifiers for each synapse in the projection.
indices	Optional item for indexing synapses by presynaptic and postsynaptic neurons.
source_to_target	Item describing synapses by presynaptic neurons.
target_to_source	Item describing synapses by postsynaptic neurons.
node_id_to_range	Item containing a data set with the identifiers of presynaptic neurons for each synapse in the projection.
range_to_edge_id	Item containing a data set with the identifiers of postsynaptic neurons for each synapse in the projection.
< synapse_group >	Mandatory item containing the ID of the neuron group.
delay	Optional item containing a data set with the values of group synapse delays.
syn_weight	Optional item containing a data set with the values of group synapse weights.
dynamics_params	Optional dynamic attributes.

Data exchange mechanism in Kaspersky Neuromorphic Platform

When working with the source code, consider the characteristics of the platform data exchange mechanism.

Kaspersky Neuromorphic Platform ensures data exchange between populations and projections, and between the platform objects and the external environment objects. Populations and projections share data via messages.

Data exchange between populations and projections

The population's neurons generate messages containing spikes to which more than one projection can be subscribed. Population spikes are sent to the synapse projections.

After receiving spikes, the backend calculates synaptic impacts. Projections send messages with the values of the synaptic impacts to the postsynaptic populations of neurons.

Data exchange between the backend and the external environment

The backend receives data from the external environment in the form of spikes. Spikes can be sent to the backend synchronously (upon the user's request). To receive spikes the method that blocks the data source is called.

Messages from the backend to the external environment can be sent to the subscribers upon a signal from the backend or in synchronous mode upon the user's request.

If needed, you can implement and use asynchronous mode for data sharing between the backend and the external environment.

Data exchange between the neural network and the external environment

Data from the external environment gets into the neural network through the input channels, which convert this data to spikes.

The neural network transmits data to the external environment through the output channels connected to the specified populations that convert spikes into data. Transmission of data, such as the membrane potential, is also possible through a monitoring subsystem.

If needed, you can implement and use custom platform objects for data sharing between the neural network and the external environment.

Data formats processed by the platform

The basic Kaspersky Neuromorphic Platform data format (hereafter KNP format) is used to store input and output data as a time series. The base format file contains the following information:

- File header containing the following information:
  - File format version

    - Array of data senders' UIDs for each step of the neural network execution (optional)
    - Array of tags
- The input or output data is a time series consisting of a list of structures with the following information:
    - Neural network execution step when the spikes were generated
    - Indices of the neurons that generated spikes at this step

Using the Kaspersky Neuromorphic Platform functions, you can convert files in the basic format to `SpikeMessages`.

KNP messages are serialized in the following notations:

- `HDF5` is the main notation for storing data in a binary form
- `JSON` is a JavaScript object notation that can be used to debug a neural network

In addition to the basic format, the input data can also be stored and transmitted as CSV files, which contain the indices of the neurons that generated the spikes. CSV files with the input data can be passed to the objects of the `IndexConverter` class for further processing in the input channels.

About Kaspersky Neuromorphic Platform backend

The platform provides several independent backends for execution of spiking neural networks on various computing devices.

The Kaspersky Neuromorphic Platform backends perform the following functions:

- Start spiking neural networks on various computers.
- Perform static verifications of the spiking neural network design.
- Data exchange between the neural network and the external environment.

You can use the following platform components to implement backends:

- *core component*. Provides classes, such as projection and population classes, required for interacting with a backend. The component is supported for C++ and Python.
- *devices component*. Provides a set of interfaces for interacting with physical devices. The component is supported for C++.
- *neuron_traits component*. Contains neuron properties. The component is supported for C++ and Python.
- *synapse_traits component*. Contains synapse model properties. The component is supported for C++ and Python.
- *meta component*. Provides general-purpose templates. The component is supported for C++.

Do not use a framework component to implement backends to avoid circular dependencies between the framework and backends.

Supported backends

Kaspersky Neuromorphic Platform supports the following types of backends:

- CPU backends:
  - *Single-thread CPU backend*. The backend works in one thread only. You can use a single-thread backend for debugging the platform and neural network operation. The interface to the backend is implemented in the `SingleThreadedCPUBackend` class.
  - *Multi-thread CPU backend*. The backend works in several threads simultaneously, thus increasing the speed of the neural network calculation. You can use a multi-thread backend to train and execute a spiking neural network. The interface to the backend is implemented in the `MultiThreadedCPUBackend` class.
- Backend for execution of spiking neural networks on an AltAI-1 neuromorphic processor (AltAI SNN backend):

  The interface to the backend is implemented in the `AltAIBackend` class. The `AltAIBackend` class provides the standard functions of the `Backend` base class.

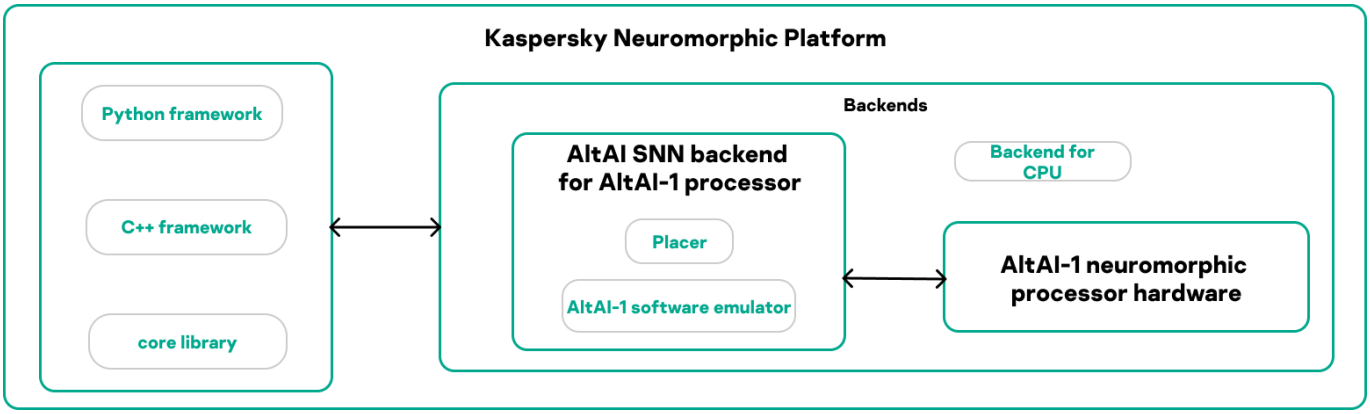
  You can select the hardware processor of the AltAI-1 neuromorphic processor or a software emulator as the device type. The hardware processor unit uses a fixed core configuration. If selecting a software emulator as the device type, you can change the configuration settings of the AltAI-1 neuromorphic processing unit cores by using the `load_core_grid_params()` software emulator method, and set the location of the input and output ports in the core configuration by using the `make_standart_north_port()` method.

  > Call `load_core_grid_params` method before initializing the backend.

  The figure below shows a diagram of the interaction of platform components when using the `AltAIBackend` backend. A spiking neural network is created with platform components. Train the neural network using the synaptic plasticity method. `AltAIBackend` currently supports neurons based on the `AltAILIF` model and synapses based on the `DeltaSynapse` model. Synapse weights must be integers in the range from `-255` to `255`.

  > Training of spiking neural network is available on a central processor using the synaptic plasticity method. If needed, you can implement and use custom training methods and backends.

  The neural network is then validated against the constraints of the neuromorphic processor AltAI-1 and translated into a format that can be executed by the processing unit. You can execute a spiking neural network both on the software emulator or directly on the AltAI-1 hardware processor.



Scheme of interaction between the platform and the AltAI SNN backend

- Backends for training and execution of artificial neural networks converted to spiking neural networks (ANN2SNN backends):
  - *AltAI-1 neuromorphic processor hardware backend*
  - *AltAI-1 neuromorphic processor software emulator backend*

  The neural network is trained by backward propagation of the error with quantization of the weights.

  The figure below shows a diagram of the interaction of platform components when using `AltAI ANN2SNN` backends. The neural network is created using the ternary layers of the `ANN2SNN` package library, implemented as part of the Python framework, and is trained by back-propagating an error with quantization of weights. The neural network is then placed, using the `placer` utility, to be executed on the AltAI-1 neuromorphic processing unit. During placement, the neural network is validated against the constraints of the neuromorphic processing unit and translated into a format that can be executed by the processing unit.

  > AltAI ANN2SNN backends are implemented in a separate package from CPU and AltAI SNN backends.

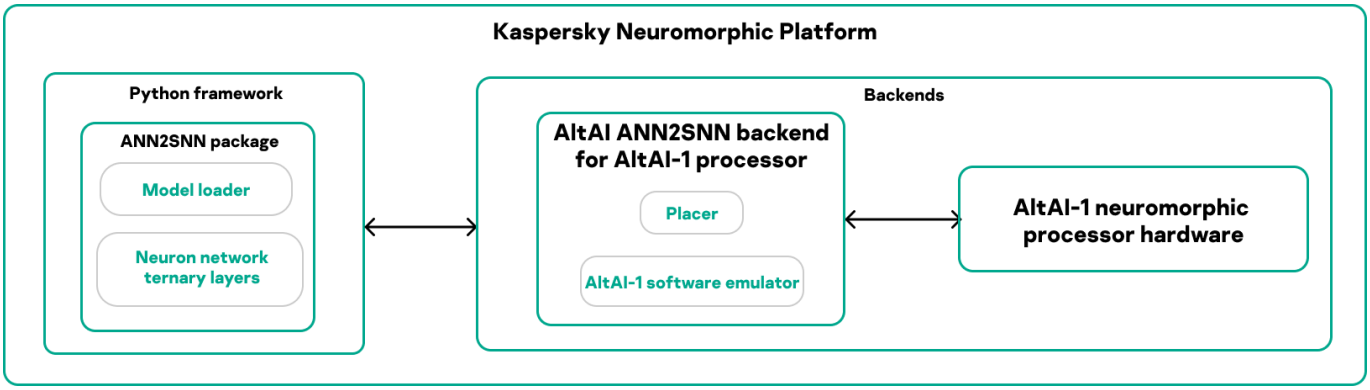


Diagram of interaction between platform components when using the AltAI ANN2SNN package

About the AltAI-1 neural processing unit

The *AltAI-1 neural processing unit* is a very large-scale integration (VLSI) circuit with a computational architecture that imitates the operation of biological sensory and nervous systems. The AltAI-1 neuromorphic processor is intended for the execution (inference) of spiking neural networks created using platform components or using ternary layers of the ANN2SNN package. The AltAI-1 architecture implements the "computation next to memory" principle, which makes it possible to avoid excessive power consumption (typical of the von Neumann architecture) for transferring data between the memory block and the computational core. The AltAI-1 neural processing unit has a low power consumption.

> To execute a spiking neural network, connect the backplane with the AltAI-1 neuromorphic processing unit to the computer via a `PCI-E` or `USB` interface.

The elementary unit of an impulse neural network is a neuron, which has a set of input signal lines (dendrites) and an output signal line (axon). The state of a neuron is described by neural potential. If, during the execution of the impulse neural network, the neuron potential exceeds the specified threshold, the neuron emits a spike along its axon and sends it to the specified target neurons. A spike can be either positive or negative. When modeling a neuron, the leak mechanism is also taken into account; the leak mechanism is intended for the relaxation of the neural potential.

The neurons of the impulse neural network are connected by synapses. Each synapse can be assigned a weight by which the neuron potential changes when a spike is received from a neuron associated with this synapse.

The main structural unit of the AltAI-1 neural processing unit architecture is a *core*. The core combines a group of neurons and memory for storing their parameters. The function of the neurons in the core is performed by a state machine that models the neuron behavior.

All neurons in a spiking neural network are synchronized with the help of a *tick* signal. After a signal is given, potentials of all neurons are sequentially updated by all cores, and spikes are emitted if needed. During a tick, the core performs 262,144 synaptic operations. The tick frequency does not exceed 2,000 Hz. One synaptic operation is performed in one clock cycle with a frequency of 600 MHz.

The AltAI-1 neuromorphic processor has a regular structure of cores in the form of a rectangular network, where each core is directly connected to its 4 neighbors. To transfer spikes between the neurons of different cores, a mechanism is implemented for routing such signals through the core network. Accordingly, when the function of a neuron is performed, the values of the neural potentials sent by the neurons from other cores can be added to the potential of the modeled neuron.

The AltAI-1 neuromorphic processor provides the ability to scale by increasing the number of VLSI chips. For questions regarding scaling, please contact Kaspersky specialists.

The table below provides general information about the AltAI-1 neuromorphic processor and its components:

General information about the AltAI-1 processor

Property		Restriction
VLSI	Crystal area	6 sq. mm
	Manufacturing technology	28 nm
	Maximum number of neuromorphic cores	256
	Number of neuromorphic cores in the prototype	16
	Maximum number of modeled neurons	131 072
	Number of modeled neurons in the prototype	8192
Core	Number of modeled neurons	512
	Number of input lines	512
	Range of neuron potential values	–32,768 to 32,767 The neuron potential is stored as a 16-bit value.
	Range of synapse weight values	The synapse weight is stored as an 8-bit unsigned value.
	Variation of weights per neuron	4

	Number of synaptic operations	9 Each synaptic connection between the neuron being modeled and the input line is characterized by one of the nine synaptic operations. You can control the impact of this synaptic connection on the neuron by selecting an operation. The following synaptic operations are available: • Empty operation. No changes are made during this operation. • Add weight N, where N is the weight number in the range 1 to 4. • Subtract weight N, where N is the weight number, in the range 1 to 4.
	Number of synaptic operations per tick	262 144
	Tick frequency	No more than 2,000 Hz
	Core power consumption	No more than 4 mW
	Core memory size	88 KB
	Core memory size for storing neuron parameters	16 KB

About CPU backend operation

The CPU backend operation is based on the message exchange between projections and populations. Objects of the `core` library can also exchange messages with the objects of the external environment. The backend provides a `MessageBus` for message exchange.

The backend operation cycle consists of the following stages:

1. Input channels send data to the `MessageBus`.
2. The backend computes the neuron population function.
3. Neuron populations send spikes with the calculation results to the `MessageBus` using `MessageEndpoints`.
4. The `MessageBus` sends data received from populations to projections or to output channels.
5. The backend computes the synapse projection function.
6. Synapse projections send synaptic impacts with the computation results to the `MessageBus` using `MessageEndpoints`.
7. The `MessageBus` sends data received from the projections to the associated neuron populations.

Installing and uninstalling the platform

This section contains step-by-step instructions on how to install and uninstall Kaspersky Neuromorphic Platform.

For the development of application solutions on Linux, you can install the platform in one of the following ways:

- Install deb packages.
- Install packages for the development of application solutions in `Python`.

When working with the platform source code on Linux or Windows, you can build the platform or an application solution using C++. The platform build can be used to develop the Kaspersky Neuromorphic Platform.

Installing deb packages

The following packages must be installed for application development:

- deb packages with CPU backend binaries
- deb package containing AltAI SNN backend binaries
- deb package with C++ framework binary
- dev-deb packages for each deb package with C++ binary code
- deb package containing a Python framework for an `AltAI SNN` backend
- deb package containing an `ANN2SNN` package and AltAI ANN2SNN backend binaries
- deb package with Kaspersky Neuromorphic Platform usage examples
- deb package with the API reference guide

The computer must have software installed as defined in the hardware and software requirements.

The deb packages are installed on Linux.

To install the deb packages:

1. To install deb packages containing CPU and `AltAI SNN` backend binaries, run the following commands as root:

   ```
   dpkg -i <path-to-deb-package>/knp-cpu-single-threaded-backend_<knp_version>_amd64.deb
   dpkg -i <path-to-deb-package>/knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb
   dpkg -i <path-to-deb-package>/knp-altai-backend_<motiv_version>_amd64.deb
   ```

   where:

   - `<path-to-deb-package>` is the path to the deb package.

- `<knp_version>` is the Kaspersky Neuromorphic Platform version.
- `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package.

When installing the `knp-altai-backend_<motiv_version>_amd64.deb` package, accept the AltAI-1 backends and `ANN2SNN` package Terms of Use. Your acceptance of the Terms of Use is a prerequisite for installing the package. The package installation process will be interrupted unless you accept the Terms of Use.

2. To install the deb package with C++ framework binary code, run the following command as root:

```
dpkg -i <path-to-deb-package>/knp-cpp-framework_<knp_version>_amd64.deb
```

3. To install dev-deb packages, run the following commands as root:

```
dpkg -i <path-to-deb-package>/knp-cpu-multi-threaded-backend-dev_<knp_version>_amd64.deb
dpkg -i <path-to-deb-package>/knp-cpu-single-threaded-backend-dev_<knp_version>_amd64.deb
dpkg -i <path-to-deb-package>/knp-altai-backend-dev_<motiv_version>_amd64.deb
dpkg -i <path-to-deb-package>/knp-cpp-framework-dev_<knp_version>_amd64.deb
```

When installing the `knp-altai-backend-dev_<motiv_version>_amd64.deb` package, accept the AltAI-1 backends and `ANN2SNN` package Terms of Use. Your acceptance of the Terms of Use is a prerequisite for installing the package. The package installation process will be interrupted unless you accept the Terms of Use.

4. To install the deb package with a Python framework for an `AltAI SNN` backend, run the following as root:

```
dpkg -i <path-to-deb-package>/knp-python3-framework_<knp_version>_amd64.deb
```

5. To install the deb packages with `ANN2SNN` package and `AltAI ANN2SNN` backend binary code, run the following commands as root:

```
dpkg -i <path-to-deb-package>/knp_ann2snn.deb
```

When installing the `knp_ann2snn.deb` package, accept the AltAI-1 backends and `ANN2SNN` package Terms of Use. Your acceptance of the Terms of Use is a prerequisite for installing the package. The package installation process will be interrupted unless you accept the Terms of Use.

6. To install the deb package with platform usage examples, run the following command as root:

```
dpkg -i <path-to-deb-package>/knp-examples_<knp_version>_amd64.deb
```

7. To install the deb package with the API reference guide, run the following command as root:

```
dpkg -i <path-to-deb-package>/knp-documentation_<knp_version>_all.deb
```

Installing Python development packages

The following DEB or WHL packages must be installed for Python development:

- deb packages:
  - `knp-python3-framework_<knp_version>_amd64.deb`: a deb package containing a Python framework for an AltAI SNN backend.
  - `knp_ann2snn.deb`: a deb package with the `ANN2SNN` package, AltAI ANN2SNN backend binary code for the AltAI-1 neuromorphic processor, and a component for placing the neural network on AltAI-1.
- whl packages for installing from PyPI:
  - `knp-<knp_version>-py3-none-any.whl`: a whl package containing a Python framework for an `AltAI SNN` backend.
  - `knp_ann2snn-<motiv_version>-py3-none-any.whl`: a whl package with the `ANN2SNN` package, `AltAI ANN2SNN` backend binary code for the AltAI-1 neuromorphic processor, and a component for placing the neural network on AltAI-1.

    This package is installed on x86-based computers.
  - `knp_ann2snn_aarch64-<motiv_version>-py3-none-any.whl`: a whl package with the `ANN2SNN` package and the `AltAI ANN2SNN` backend binary code for the AltAI-1 neuromorphic processor.

    This package is installed on ARM-based computers.

  where:
  - `<knp_version>` is the Kaspersky Neuromorphic Platform version.
  - `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package.

The packages are installed on Linux.

To install deb packages for Python development:

1. To install the deb package with a Python framework for an `AltAI SNN` backend, run the following as root:

   ```
   dpkg -i <path-to-deb-package>/knp-python3-framework_<knp_version>_amd64.deb
   ```

   where:
   - `<path-to-deb-package>` is the path to the deb package.
   - `<knp_version>` is the Kaspersky Neuromorphic Platform version.
2. To install the deb packages with `ANN2SNN` package and `AltAI ANN2SNN` backend binary code, run the following commands as root:

   ```
   dpkg -i <path-to-deb-package>/knp_ann2snn.deb
   ```

When installing the `knp_ann2snn.deb` package, accept the AltAI-1 backends and `ANN2SNN` package Terms of Use. Your acceptance of the Terms of Use is a prerequisite for installing the package. The package installation process will be interrupted unless you accept the Terms of Use.

Use the `pip3` package manager to install whl packages from the PyPI platform.

To install whl packages:

1. To install a whl package containing a Python framework for an `AltAI SNN` backend, run the following command:

   ```
   pip3 install --break-system-packages knp-<knp_version>-py3-none-any.whl
   ```

   where `<knp_version>` is the Kaspersky Neuromorphic Platform version.

2. To install a whl package containing an `ANN2SNN` package and `AltAI ANN2SNN` backend binaries, do one of the following:

   - To install a package on an x86-based computer, run:

     ```
     pip3 install --break-system-packages knp_ann2snn-<motiv_version>-py3-none-any.whl
     ```

   - If you need to install a package on an ARM-based computer, run:

     ```
     pip3 install --break-system-packages knp_ann2snn_aarch64-<motiv_version>-py3-none-any.whl
     ```

   where `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package.

Scenario for building a platform project or an application solution

You can build a platform or an application solution in C++. The platform build can be used when developing the Kaspersky Neuromorphic Platform.

When developing application solutions, you can install the required deb packages instead of building the solution.

> You can also use the Docker image `knp-build-image` included in the platform distribution kit to build the platform or an application.

A platform or application solution build script consists of the following stages:

1. **Obtaining the platform source code**

   You can get the platform source code in one of the following ways:

   - Download the archive with the source code from the platform repository and unpack it to the working directory.
   - Clone the platform repository.

2. **Defining the Boost_ROOT setting in Windows**

In Windows, define the `Boost_ROOT` setting. To do this, create an environment variable or specify the path to the installed Boost library in the `CMakePresets.json` file located in the root directory with the platform source code.

You can define the path to the installed Boost library as a CMake invocation parameter using the following command:

```
cmake -DBOOST-ROOT=<path to installed Boost library>
```

3. **Configuring the build settings for the application solution**

   In case of an application solution build, describe the build process in the `CMakeLists.txt` file. Specify the directory with the application to be built using the `add_subdirectory` function, specify the executable file using the `add_executable` function, and specify the platform libraries to be connected to the project using the `target_link_libraries` function. You can use the following libraries:

   - `neuron_traits`
   - `synapse_traits`
   - `core`
   - `meta`
   - `devices`
   - `backends`
   - `framework`

4. **Configuring a platform or an application solution build**

   If you are building a platform or an application solution on Linux, configure the build using the CMake build system. For more details on configuring a build using CMake, refer to the CMake documentation.

   If you are building the platform or application solution in Windows, configure the build in Visual Studio. To do this, open the platform or application project by selecting the required `CMakeLists.txt` file and configure the cache. In case of an application solution build, select the `CMakeLists.txt` file of the application solution. In case of a platform build, select the `CMakeLists.txt` file located in the root directory with the platform source code.

   > The first build configuration with CMake may take too long to complete and fail with network errors. If network errors occur, please run the build configuration again.

5. **Starting the build**

   Start the platform or application solution build. For more details on starting the build of projects, refer to the CMake documentation or the documentation of the integrated development environment (IDE) being used.

Downloading and unpacking the platform source code archive

To download and unpack the platform source code archive:

1. Download the archive with the source code from the platform repository.

2. Unpack the platform archive to the working directory.

   For example, you can unpack the platform archive using the following command:

```
tar xf <path_to_platform_archive> --directory <path_to_working_directory>
```

Defining the Boost_ROOT setting in Windows

Before building a platform or an application solution on Windows, define the `Boost_ROOT` setting. As the setting value, specify the path to the installed Boost library. To do this, specify it as the value of the `Boost_ROOT` setting in the `CMakePresets.json` file located in the root directory with the platform source code.

You can also create an environment variable for the `Boost_ROOT` setting.

To create an environment variable in Windows:

1. In Windows, press the **Windows**+**R** key combination.
2. Type `sysdm.cpl` and press **Enter**.
3. In the window that opens, select the **Additional** tab and click the **Environment variables** button.
4. In the window that opens, create the `Boost_ROOT` variable and specify the path to the Boost library as the value of the variable.

Configuring the project build settings for the application solution in C++

To build an application solution, specify the build settings in the `CMakeLists.txt` file. The `CMakeLists.txt` file must contain the following commands:

cmake_minimum_required(VERSION 3.25)

Specifies the minimum supported version of CMake. To build a project based on the Kaspersky Neuromorphic Platform, CMake version 3.25 or later is required.

project (<project_name>)

Specifies the project name.

set(CMAKE_CXX_STANDARD 17)

Specifies C++ 17 as the locale to use.

add_subdirectory (<platform_source_code_directory>)

Specifies the directory containing the application to be built.

add_executable("${PROJECT_NAME}" <application_file>)

Specifies the executable file.

target_link_libraries("${PROJECT_NAME}" PRIVATE <platform_link_libraries>)

Specifies the Kaspersky Neuromorphic Platform libraries to be linked to your project. You can connect the following libraries: `backends`, `neuron_trits`, `synapse_traits`, `core`, `meta`, `devices`, and `framework`.

Before building a project in Windows, define the `Boost_ROOT` setting.

To configure the project build settings for the application solution in C++:

1. Navigate to the working directory.
2. In the working directory, create the `CMakeLists.txt` file.
3. Open the created file in any text editor and specify the project build settings.

   Example of the CMakeLists.txt file

   ```
   cmake_minimum_required(VERSION 3.25)
   # digits-recognition is a project name.
   project(digits-recognition)

   set(CMAKE_CXX_STANDARD 17)

   # Runs CMake in the KNP directory and adds to the build all projects from the directory.
   add_subdirectory(KNP)

   # Adds a binary file with the source code to the main.cpp application file.
   add_executable("${PROJECT_NAME}" main.cpp)
   # Connects the framework and the single-thread CPU backend library to the digits-recognition project.
   target_link_libraries("${PROJECT_NAME}" PRIVATE KNP::BaseFramework::Core KNP::Backends::CPUSingleThreaded)
   ```

   If needed, you can load the framework dynamically.
4. Save your changes to the `CMakeLists.txt` file.

Configuring a build of a platform or an application solution in Windows

In Windows, you can configure how your platform or application solution is built using Visual Studio.

To configure a build of a platform or an application solution in Windows:

1. In Windows, start Visual Studio.
2. On the **File** tab, select **Open** → **CMake**.
3. In the window that opens, select the root `CMakeLists.txt` file.

   In case of an application solution build, select the `CMakeLists.txt` file of the application solution. In case of a platform build, select the `CMakeLists.txt` file located in the root directory with the platform source code.
4. On the **Project** tab, select **Configure cache**.

   CMake will begin working on the build configuration.

After CMake finishes its configuration work, start the build.

Uninstalling the platform

Depending on the installation method, you can uninstall the platform in the following ways:

- Uninstall the deb packages.
- Uninstall the packages for the development of application solutions in Python.

- Delete the directory into which the platform source code archive was unpacked.

Uninstalling the deb packages

We recommend using the `apt` package manager to uninstall deb packages.

To uninstall the deb packages:

1. To uninstall deb packages containing CPU and `AltAI SNN` backend binaries, run the following commands as root:

```
apt purge knp-cpu-single-threaded-backend
apt purge knp-cpu-multi-threaded-backend
apt purge knp-altai-backend
```

2. To uninstall the deb package with C++ framework binary code, run the following command as root:

```
apt purge knp-cpp-framework
```

3. To uninstall the dev-deb packages, run the following commands as root:

```
apt purge knp-cpu-multi-threaded-backend-dev
apt purge knp-cpu-single-threaded-backend-dev
apt purge knp-altai-backend-dev
apt purge knp-cpp-framework-dev
```

4. To uninstall a deb package with a Python framework for an `AltAI SNN` backend, run the following as root:

```
apt purge knp-python3-framework
```

5. To remove a deb package containing an `ANN2SNN` package and `AltAI ANN2SNN` backend binaries, run the following command:

```
apt purge knp_ann2snn
```

6. To uninstall the deb package with platform usage examples, run the following command as root:

```
apt purge knp-examples
```

7. To uninstall the deb package with the API reference guide, run the following command as root:

```
apt purge knp-documentation
```

Uninstalling the packages for the development of application solutions in Python

To remove deb packages used for Python development, we recommend using the `apt` package manager.

To uninstall the deb packages:

1. To uninstall a deb package with a Python framework for an `AltAI SNN` backend, run the following as root:

   ```
   apt purge knp-python3-framework
   ```

2. To uninstall the deb package with `ANN2SNN` package and `AltAI ANN2SNN` backend binary code, run the following commands as root:

   ```
   apt purge knp_ann2snn
   ```

To uninstall a whl package, use the `pip3` package manager.

To uninstall whl packages:

1. To uninstall a whl package containing a Python framework for an `AltAI SNN` backend, run the following command:

   ```
   pip3 uninstall knp-<knp_version>-py3-none-any.whl
   ```

   where `<knp_version>` is the Kaspersky Neuromorphic Platform version.

2. To uninstall a whl package containing an `ANN2SNN` package and `AltAI ANN2SNN` backend binaries, do one of the following:

   - To uninstall a package from an x86-based computer, run:

     ```
     pip3 uninstall knp_ann2snn-<motiv_version>-py3-none-any.whl
     ```

   - To uninstall a package from an ARM-based computer, run:

     ```
     pip3 uninstall knp_ann2snn_aarch64-<motiv_version>-py3-none-any.whl
     ```

   where `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package.

Removing the platform source code

To remove the platform source code,

delete the directory to which the platform source code archive was unpacked.

C++ platform components

This section describes the Kaspersky Neuromorphic Platform components written in C++.

neuron_traits library

The neuron `traits library` is a set of neuron attributes. You can use the neuron attributes provided in the library, for example, when constructing populations.

Currently, the library supports the property sets for the following neurons:

- Neuron based on the `AltAILIF` model

  The neuron based on the `AltAILIF` model is a leaky integrator neuron (also known as a leaky integrate-and-fire neuron). The neuron is characterized by the membrane potential and the threshold values of the membrane potential.

  The value of the membrane potential changes in accordance with the value of the synaptic impact coming from the associated synapse. If the membrane potential of a neuron reaches a positive or negative threshold value, the neuron generates a positive or negative spike, respectively. Depending on the specified attributes of the neuron, after one of the thresholds is exceeded, the membrane potential either takes on the value by which the threshold was exceeded, or is reset to a specified value.

  The value of the membrane potential also changes linearly by the leakage value `potential_leak_`, which can be either positive or negative.

  In the attributes of a neuron instance, you can specify the baseline value of the membrane potential, the value of the membrane potential after the reset, a positive and negative threshold value of the potential, and the leakage value; you can also set flags that control the behavior of the potential after the thresholds are exceeded, a flag for resetting the neuron potential at the beginning of each time step, and a flag controlling the leakage behavior depending on the current neuron potential sign.

- Neuron based on the `BLIFATNeuron` model ("BLIFAT neuron").

  A neuron based on the `BLIFATNeuron` model is an integrator neuron with a leak and an adaptive threshold, which generates a series of spikes (Bursting Leaky Integrate-and-Fire with Adaptive Threshold Neuron). A BLIFAT neuron is characterized by the membrane potential and the threshold value of the membrane potential.

  The value of the membrane potential changes in accordance with the value of the synaptic impact coming from the associated synapse. If the neuron membrane potential reaches a threshold value, the neuron starts generating a series of spikes with a certain interval. In this case, after the spike generation, the membrane potential tends to the base value, and the threshold value changes in accordance with the increment.

  If a neuron does not receive messages from a connected synapse for a certain time, the membrane potential and its threshold value exponentially tend to the base values.

  You can specify the base value of the membrane potential, the base threshold value, the increment value, the time interval upon reaching which the values of the membrane potential and potential threshold begin to approach the base value, the duration of a spike series, and the interval for spike generation in the attributes of a neuron instance.

- Neuron based on the `SynapticResourceSTDPBLIFATNeuron` model.

  A neuron based on the `SynapticResource STDPBLIFATNeuron` model is a BLIFAT neuron characterized by a free synaptic resource, a threshold value of a free synaptic resource, and stability.

  If the absolute value of the free synaptic resource exceeds the threshold value, this free synaptic resource is distributed equally among all plastic presynaptic connections of the neuron. After that, the value of the free synaptic resource of the neuron equals `0`. If the absolute value of the free synaptic resource is less than the threshold value, the neuron appropriates the synaptic resource from connected synapses until it reaches its threshold value.

The neuron stability value changes in accordance with the synaptic plasticity used by the synapse and affects its plasticity value.

For more details on the properties of neurons, see API documentation for Kaspersky Neuromorphic Platform. If necessary, you can implement and use your own neuron models when working with the source code.

synapse_traits library

The `synapse_traits` library is a set of synapse attributes. You can use the synapses provided in the library, for example, when creating projections.

Currently, the library supports the property sets for the following synapses:

- Synapse based on the `DeltaSynapse` model (hereinafter also referred to as delta synapse).

  After receiving spikes from a presynaptic population, the synapses generate a message containing the synaptic impact values. The synapse projection sends the synaptic impact to the population to change the membrane potential of the postsynaptic neuron. In this case, the message is sent by the projection without a time delay.

  The sent synaptic impact is determined by the weight of the delta synapse. You can specify the synapse weight in the attributes of its instance.

- Synapse based on the `AdditiveSTDPDeltaSynapse` model.

  A synapse based on the `AdditiveSTDPDeltaSynapse` model is a delta synapse whose synaptic plasticity depends on the time of spike receipt. If a spike from the presynaptic population is received by the projection before the postsynaptic neuron generates the presynaptic spike, the synapse weight value increases. If a presynaptic spike is received by the synapse after the postsynaptic neuron generates a spike, the weight of the synapse decreases.

  The synapse weight determines the value of the synaptic impact sent to the postsynaptic population.

  The change in synapse weight is calculated using the formula shown below, where:

  - *j* corresponds to a presynaptic neuron.
  - *i* corresponds to a postsynaptic neuron.
  - *W* corresponds to the synaptic plasticity function.
  - *n* corresponds to the sequential number of the presynaptic neuron.
  - *f* corresponds to the sequential number of the postsynaptic neuron.
  - t_i^n corresponds to the time when the synapse receives spikes from a presynaptic neuron.
  - t_j^f corresponds to the time when the postsynaptic neuron generates the spike.
  - Δw_j corresponds to the value of the change in synapse weight after a spike is received from a presynaptic neuron.

$$\Delta w_j = \sum_{f=1}^{N} \sum_{n=1}^{N} W(t_i^n - t_j^f)$$

Formula for calculating the change in synapse weight based on the AdditiveSTDPDeltaSynapse model

- Synapse based on the `SynapticResourceSTDPDeltaSynapse` model.

  A synapse based on the `SynapticResourceSTDPDeltaSynapse` model is a delta synapse whose synaptic plasticity depends on the synaptic resource. The value of the synaptic resource depends on the synaptic plasticity:

  - Hebbian plasticity.

    The value of a synaptic resource depends on the period of the dense sequence of spikes (hereinafter also referred to as DSS). *DSS* is a sequence of unforced spikes generated by a neuron, where all adjacent spikes are separated by the time intervals by no more than *ISImax*, where *ISImax* is the maximum allowed distance between spikes in the DSS. The *DSS period* is the time interval between the moment when the Hebbian plasticity starts before the first spike in the DSS and one of the following moments: when the maximum allowable distance ISImax is reached after the last spike in the DSS, when a forced activation occurs, or when the spike arrives at the dopamine synapse.

    When at least one spike is received within the DSS period, the synaptic resource changes by the value of the current Hebbian plasticity. The Hebbian plasticity depends on the neuron stability variable and is calculated using the formula shown below, where:

    - d_H corresponds to Hebbian plasticity.
    - *s* corresponds to the neuron stability variable. At the time of the first spike within the DSS period, the stability value changes by a constant. If the stability is negative, it does not change downward.

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

    Formula for calculating Hebbian plasticity

  - Unconditional plasticity.

    In the case of a nonzero unconditioned plasticity, the value of the synaptic resource decreases by the value of plasticity when a spike is received. The unconditional plasticity depends on the neuron stability variable and is calculated using the formula shown below, where:

    - d_H corresponds to unconditional plasticity.
    - *s* corresponds to the neuron stability variable. At the time of the first spike within the DSS period, the stability value changes by a constant. If the stability is negative, it does not change downward.

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

    Formula for calculating unconditional plasticity

  - Dopamine plasticity.

    The dopamine plasticity is applicable when the sum of all dopamine synapses that receive a spike at the current step of the neural network execution is not zero. In the case of dopamine plasticity, the value of the synaptic resource changes based on the formula below, where:

    - *D* corresponds to the sum of dopamine synapses.
    - *s* corresponds to the neuron stability variable.

    $$D\min(2^{-s}, 1)$$

    Formula for calculating synaptic resource in case of dopamine plasticity

  The sum of dopamine synapses affects the neuron stability variable. If at the time of the forced activation, the *dopamine punishment* is received (the sum of dopamine synapses is less than zero), the value of the neuron stability is decreased by the *rD* value, where *r* is a constant and *D* is dopamine punishment.

If at the time of the forced activation, the *dopamine reward* is received (the sum of dopamine synapses is greater than zero), the value of the neuron stability is increased by the *rD* value, where *r* is a constant and *D* is dopamine reward. Otherwise, the stability value changes to the value shown below, where:

- *D* corresponds to the sum of dopamine synapses.
- *ISImax* corresponds to the maximum acceptable distance between spikes in the DSS.
- t_{TSS} corresponds to the time since the first spike in the last DSS.

$$D\max\left(2 - \frac{|t_{TSS} - ISI_{max}|}{ISI_{max}}, -1\right)$$

Formula for calculating the neuron stability change in case of the dopamine reward

The synapse weight determines the value of the synaptic impact sent to the postsynaptic population and is associated with the value of the synaptic resource by the formula shown below, where:

- *w* corresponds to the synapse weight.
- *W* corresponds to a synaptic resource.
- $\boldsymbol{w}_{\min}$ corresponds to a random minimum weight.
- $\boldsymbol{w}_{max}$ corresponds to a random maximum weight.

$$\boldsymbol{w} = \boldsymbol{w}_{\min} + \frac{(\boldsymbol{w}_{max} - \boldsymbol{w}_{min}) * max(W, 0)}{\boldsymbol{w}_{max} - \boldsymbol{w}_{min} + max(W, 0)}$$

Formula for calculating the weight change depending on the synaptic resource

For more details on the properties of synapses, see API documentation for Kaspersky Neuromorphic Platform. If needed, you can implement and use your own synapse models when working with source code.

core library

The `core` library provides support for backends. The `core` library provides the classes required for interacting with backends and performing application tasks.

The table below contains descriptions of the library components.

The core library components

Component	Description
`Backend`	Base class for specific backend implementations.
`BaseData`	Structure defining a set of base data.
`continuously_uid_generator`	Class implementing a generator of unique IDs.
`Device`	Base class of devices.
`MessageBus`	Class implementing an interface for the message bus.
`MessageEndpoint`	Class implementing an interface for the message bus endpoint.
`Population`	Template class that implements the container of neurons of the same model.
`Projection`	Template class that implements the container of synapses of the same type.
`Subscription`	Class implementing the message subscription.
`TagMap`	Class implementing an interface for the tag dictionary.

`UID`	Class defining unique identifiers.
`uid_hash`	Function implementing hashing of UIDs.
`messaging`	Namespace that contains message interfaces.
`tags`	A namespace that contains the names and value types of known tags.

The figure below shows how the library components interact.

In the Kaspersky Neuromorphic Platform, the `BaseData` structure contains basic information about an object and includes `UID` and `TagMap`. The `BaseData` structure is used in the following classes: `Projection`, `Population`, and `Backend`. Therefore, each of them has a `UID` and a set of tags with the `TagMap` values.

The `Population` and `Projection` classes receive and send messages that are implemented in the classes of the `messaging` namespace. Messages contain the sender's UID in the header.

The `Backend` class contains the `Device` and `MessageBus` classes. The `MessageBus` creates `MessageEndpoint` and interacts with them by exchanging messages. Each `MessageEndpoint` has a container of the `Subscription` class.

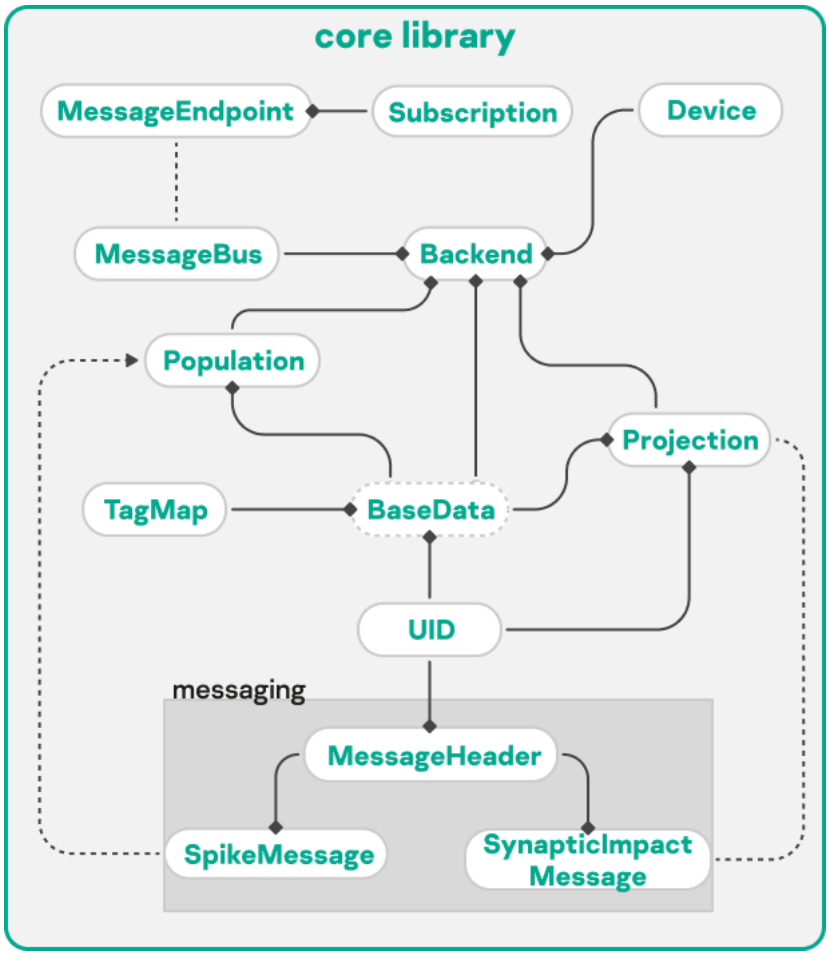


Diagram of interactions between components in the core library

UID class

An object of the `UID` class is part of the `BaseData` structure; it represents the identifier of the object. The identifier is unique.

uid_hash function

The `uid_hash` function implements the object ID hashing function.

continuously_uid_generator class

The `continuously_uid_generator` class implements a unique ID generator (UID). The ID is displayed as a string divided into groups by means of dashes and is a 128-bit number. IDs are generated sequentially in increments of 1.

You can use the `continuously_uid_generator` class to debug your platform and execute the neural network.

TagMap class

An object of the `TagMap` class is a part of the `BaseData` structure. It is an associative array that includes zero or more "tag key-tag value" records, where the tag name is the key. Tag names are unique strings.

BaseData structure

The `BaseData` structure defines a set of base data common to multiple `core` library objects and framework objects. For example, `BaseData` is used in objects of the `Backend`, `Device`, `Population`, and `Projection` classes.

`BaseData` includes the following attributes:

- `uid_`—unique ID of the object.
- `tags_`—associative `TagMap` array consisting of the tags used by the object and their values.

Population class

The `Population` template class implements a container that contains neurons belonging to the same model. When you use the template of the `Population` class, you can specify one of the neuron types supported by the `neuron_traits` library.

Using the `Population` class, you can send the population data to the backend for computing their attributes. After computation, the new population attributes are stored in the backend. To pass new attributes from the backend to the corresponding population objects, call the synchronization. If you manually change the population attributes before calling synchronization, you can pass the population data to the backend again to compute new attributes. For more information on population and backend functions, see API documentation for Kaspersky Neuromorphic Platform.

Each neuron population has the following attributes:

- Basic population data.
- Attributes of the population neurons.

Projection class

The `Projection` template class implements a container that contains synapses of one of the types supported by the `synapse_traits` library.

An object of the `Projection` class is a set of neuron connections of the presynaptic and postsynaptic populations. The projection is constructed using the connection generator.

Using the `Projection` class, you can change synapses settings within a projection, as well as send projection data to the backend and receive them from the backend.

Kaspersky Neuromorphic Platform performs data exchange between the neuron populations and the synapse projections by means of messages. The synaptic projection receives messages with the spikes from the presynaptic population and sends messages with synaptic impacts to the postsynaptic population.

Each synapse projection has the following attributes:

- Basic projection data.
- Unique ID of the presynaptic population.
- Unique ID of the postsynaptic population.
- A `boolean` value indicating if the synapse weight can be changed. If the synapse weight cannot be changed, the attribute is set to `true`, otherwise, its value is `false`.
- Synapse group attributes.
- A registry of projection synapse connections to presynaptic and postsynaptic neurons.

  The registry update depends on the projection function behavior and can be performed automatically after any projection update (for example, removing a synapse from the projection) or when the projection is used for computation.
- A `boolean` value indicating if the connection registry is up to date. If the connection registry is up to date, the attribute is set to `true`, otherwise, its value is `false`.
- Attributes used by synapses of the same type.

Backend class

The `Backend` class is the base class for backend implementations. Each backend is implemented as a library, allowing you to:

- Run training and execution of the spiking neural network on various devices
- Read output spikes received from the computer
- Get internal data to monitor the neural network operation
- Upload internal data of the neural network operation monitoring received from the computer

To initialize the backend, first download projections containing synapses and populations containing neurons. During initialization of the backend, the loaded synapse projections are subscribed to the spikes of the associated populations or input channels, and the downloaded populations are subscribed to synaptic impacts of the associated projections.

Each object of the `Backend` class has the following attributes:

- Basic backend data.
- Message bus. Using the message bus, the backend ensures the message exchange between the objects of the `core` library, as well as sending messages from modifiers to neurons and synapses.
- Message bus endpoint.
- Backend initialization status.
- Neural network execution status on the backend.
- Backend devices.
- Neural network execution step number.

Device class

The `Device` class is the base class for the device classes supported by the `devices` library.

The class inheritors override functions that you can use to get general information about the objects of the inherited classes. For example, you can get information about the type or power consumption of a specific device. The `Device` class receives data about devices from the backend.

Each object of the `Device` class contains basic data for a specific device.

MessageBus class

The `MessageBus` class implements an interface for accessing the message bus for receiving, distributing, and delivering messages between objects connected to it An object of the `MessageBus` class provides many message endpoints through which the objects of the `core` library can subscribe to the messages or send them.

The `MessageBus` class provides functions that return the message endpoints being used. If you delete a message endpoint, message transmission through this message endpoint stops.

Using a message bus allows you to implement the publisher-subscriber principle. The publisher sends messages through a message bus as a proxy. The subscription object filters messages and sends them to the subscribers. This approach allows you to avoid creating a large number of IPC channels because the publisher and subscribers are not connected directly.

MessageEndpoint class

The `MessageEndpoint` class implements an interface for accessing a message endpoint that subscribes to the messages and sends them using the message bus.

A `MessageEndpoint` object stores a set of `Subscription` objects used for filtering messages. A message bus can have multiple message endpoints.

Using the functions of the `MessageEndpoint` class, you can configure the exchange of messages between the backend and the framework objects. To send messages, first add a subscription object to the message endpoint object.

If you delete a message endpoint, the associated subscription objects are also deleted. Message transmission through this message endpoint stops.

Subscription class

The `Subscription` class implements the functionality of subscription to messages. A subscription object is used to store messages of a specific type until they are requested by the recipient. The subscription object defines the recipients of stored messages.

Subscriptions are stored at message endpoints to a message bus. If the message endpoint object that stores the subscription object is deleted, the subscription object is also deleted. Messages defined by this subscription stop being transmitted.

Each subscription object has the following attributes:

- The vector of messages to pass to the recipient.
- Message recipient ID.
- Message senders IDs.

tags namespace

The `tags` namespace implements the names and value types of known tags.

> The names and value types of known tags are implemented in the `tags` namespace to prevent a circular dependency between the backend and the C++ framework.

messaging namespace

The `messaging` namespace implements interfaces for accessing messages. The message contains data sent and received by the backend objects, as well as data sent by the objects identified by UID. Message exchange is provided by the `MessageBus` message bus.

Each message has the following attributes:

- `MessageHeader`—message header containing the following information:
  - `sender_uid_`—ID of the object that sends the message.
  - `send_time_`—the time when the message was sent.

- The message body, consisting of the set of transmitted attributes. The set of the transmitted attributes depends on the message type.

Messages are used to update user structures and synchronize changes on the backend. The table below contains a description of the supported backend message types.

Message structures

Message type	Description	Message structure
`SpikeMessage`	A message from a neuron population to a synapse projection containing a spike. More than one projection can be subscribed to the messages containing spikes.	`header_`—message header containing the UID of the population whose neurons generated the spikes and the time when the spikes were generated. `neuron_indexes_`—indices of the neurons in the population that generated spikes. Neuron indices are unique within a population.
`SynapticImpactMessage`	A message from a synapse projection to the neuron population containing synaptic impact.	`header_`—message header containing the projection UID and the time when the message was sent. `postsynaptic_population_uid_`—UID of the postsynaptic population. `Presynaptic_population_uid_`—UID of the presynaptic population. `is_forcing_`—`boolean` value indicating the use of plasticity in the synapse projection. If the synapse projection uses synaptic plasticity, the attribute is `true`, otherwise it is `false`. `impacts_`—vector containing the following attributes: • `connection_index_`—connection index. • `impact_value_`—synaptic impact value. • `synapse_type_`—type of synapse. • `presynaptic_neuron_index_`—index of the presynaptic neuron from which the spike was received by the projection synapse. The neuron index is unique within the population. • `postsynaptic_neuron_index_`—postsynaptic neuron index. The value of the synaptic impact is sent to this neuron to change the values of its attributes. The neuron index is unique within the population. The `impacts_` vector may contain more than one set of the listed attributes to convey synaptic impacts to multiple neurons in the same population.

meta library

The `meta` library is a collection of general-purpose templates. Library templates are used, for example, in the backend functions that provide introspection.

devices library

The `devices` library implements and provides a set of interfaces that you can use to interact with physical devices.

Library functions are used in backend functions to obtain information about devices. To select a device for executing the neural network, first call the backend function that provides computer types. The computer types are `Device` class inheritors with their own set of functions and properties. To select a device from those provided, call the `select_devices()` function by passing to it unique device IDs or pointers to `Device` instances.

You can obtain information about the device (for example, about power consumption) using the methods of the object that provides the interface to the device.

You can implement and use your own devices when working with source code if needed. To do this, implement a device class within the backend, provide an interface for the device, and implement a device loading function that returns a pointer to an instance of the `Device` class.

The Kaspersky Neuromorphic Platform currently has interfaces for the central processor and the AltAI-1 neuromorphic processing unit. When using an AltAI-1 neuromorphic processing unit, you can choose between a hardware processor and a software emulator as the device type. The hardware processor unit uses a fixed core configuration. If selecting a software emulator as the device type, you can change the configuration settings of the AltAI-1 neuromorphic processing unit cores by using the `load_core_grid_params()` method, and set the location of the input and output ports in the core configuration by using the `make_standart_north_port()` method.

C++ framework

C ++ framework is designed for creating spiking neural networks and training these with synaptic plasticity methods. Resulting neural networks can be executed on a central processor, an AltAI-1 neuromorphic processing unit, or a software emulator of the latter.

C ++ framework is a set of independent units that manage backends, transform the neural network representation into various formats, and select the optimal neural network to run. In the C++ framework, the functions that are not directly related to training and execution of spiking neural networks are implemented.

Using the C++ framework for Kaspersky Neuromorphic Platform, you can perform the following tasks:

- Load the neural network structure from external formats.
- Dynamically load various backends.
- Train and execute spiking neural networks.
- Input and output neural network data.
- Monitor training and execution of spiking neural networks.
- Implement special algorithms.

The C++ framework is implemented within the `framework` namespace. The table below contains descriptions of the C++ framework components.

C++ framework components

Component		Description
`coordinates`		A namespace that contains the spatial coordinates of the neuron.
`io`		A namespace that contains the `input`, `output`, and `storage` namespaces.
	`input`	A namespace that contains interfaces for accessing the input channel and the converter for data coming from the external environment.
	`output`	A namespace that contains interfaces for accessing the output channels.
	`storage`	A namespace that contains functions for loading and saving `SpikeMessage` messages.
`monitoring`		The namespace that contains the interface to the observer.

`population`	A namespace that contains functions for generating populations of neurons.
`projection`	A namespace that contains functions for generating synapse projections and generating synapses with default attribute values.
`sonata`	A namespace that contains functions to load and save the neural network in the `SONATA` format.
`synchronization`	A namespace that contains a function for getting the current state of the neural network from the backend.
`BackendLoader`	A class that implements a dynamic library loader.
`Model`	A class that implements the model.
`ModelExecutor`	A class that implements the model executor.
`ModelLoader`	A class that implements the model loader.
`Network`	A class that implements a neural network object.

The figure below shows how the C++ framework components interact.

In Kaspersky Neuromorphic Platform, a `Network` object can be loaded via the `sonata` namespace function. You also can define the way a `Network` object is displayed with a set of (arbitrary or user-defined) coordinates from the `coordinates` namespace. The state of the `Network` object can be updated from the `Backend` with the `synchronization` namespace function.

An object of the `Network` class, as well as the input and output channel containers implemented in the `input` and `output` namespaces, respectively, are stored in an object of the `Model` class. The input and output channels are used to convert data received by the neural network projections, as well as to convert messages from the neural network populations into one of the provided data formats. `SpikeMessage` messages are received in input channels and unloaded from output channels with `storage` namespace functions.

You can execute an object of the `Model` class using the `start()` function of the `ModelExecutor` class object. An object of the `ModelExecutor` class also contains an object of the `MessageObserver` class from the `monitoring` namespace, which processes the received messages in accordance with the specified function, and an object of the `ModelLoader` class that have the functions you can use to load model to the exact .backend instance. To execute the model in the backend instance, a pointer to the backend instance received from the `BackendLoader` class object is passed to the `ModelExecutor` class object.

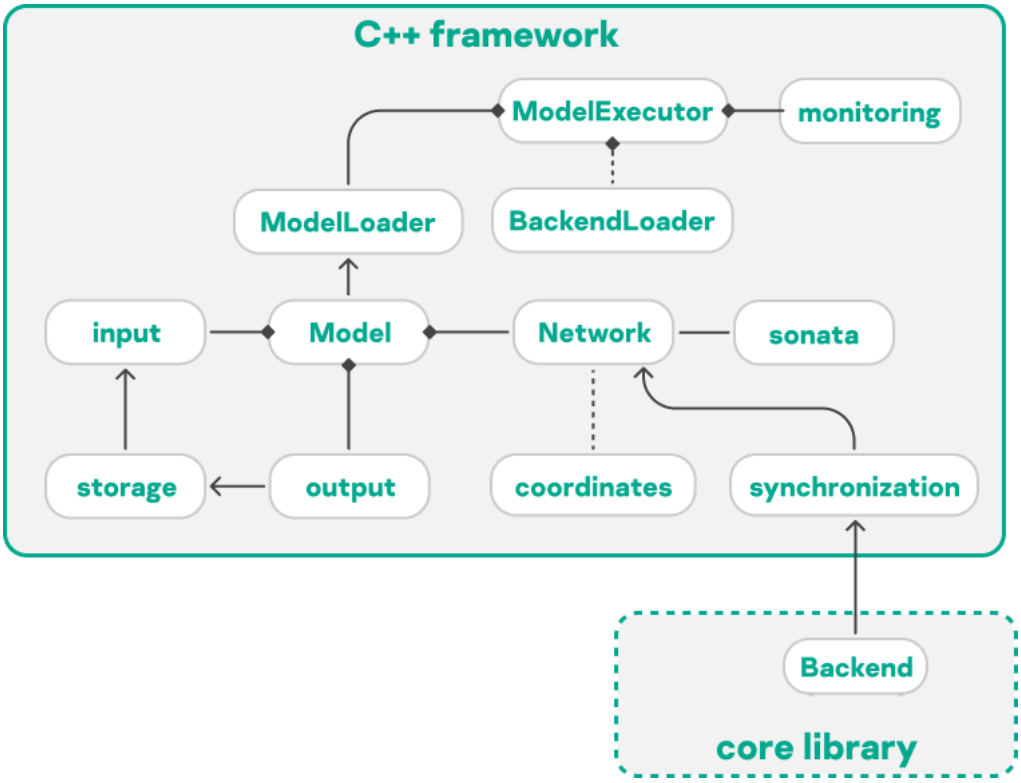


Diagram of interactions between the C++ framework components

sonata namespace

The `sonata` namespace implements functions for loading and saving a neural network in the `SONATA` format. The namespace provides the following functions:

- `load_network ()`: load the neural network. The neural network topology must be described in the `SONATA` format.
- `save_network ()`: save a neural network topology in the `SONATA` format.

coordinates namespace

`coordinates` namespace implements the coordinates of a point in an n-dimensional space. The coordinates are stored in a tag object.

The coordinates can be used to display a neural network. You can also use the coordinates to define the neural network attributes. For example, based on the distance between the neuron coordinates, you can determine the delay in the signal transmitted between these neurons.

The coordinates can change arbitrarily. The C++ framework implements a Cartesian coordinate system for two-dimensional and three-dimensional space, as well as polar and spherical coordinate systems.

If necessary, you can implement and use your own coordinate systems.

synchronization namespace

The `synchronization` namespace implements the `get_network_copy ()` function for getting the current state of the neural network from the backend. For instance, you can use this function to save neural network parameters whenever there is a shift in the accuracy of its predictions or at regular intervals during execution. You can compare the quality of the saved neural network copies to select the best one for downloading to the device.

storage namespace

The `storage` namespace contains the `native` namespace, which implements functions for loading and saving `SpikeMessage` messages. The namespace provides the following functions:

- `load_messages_from_json`: loads messages from a JSON file.
- `load_messages_from_h5`: loads messages from an HDF5 file.
- `save_messages_to_json`: saves messages to a JSON file.
- `save_messages_to_h5`: saves messages to an HDF5 file.

population namespace

The `population` namespace contains the `creators` and `neurons_generators` namespaces.

creators namespace

The creators namespace implements functions for generating neuron populations. The namespace provides the following functions:

- `from_container` to generate a population of neurons of a certain type from a container containing the neuron attributes.
- `make_random` to generate a population of neurons of a certain type with random attribute values.
- `make_default` to generate a population of neurons of a certain type with default attribute values.
- `make_clone` to generate a population of neurons of a certain type whose attributes are copied from those of a given neuron.

neurons_generators namespace

The `neurons_generators` namespace implements functions and interfaces for generating neurons. The namespace provides the following functions and interfaces:

- The `from_container` function to generate a population of neurons of a certain type from a container with neuron attributes.
- The `MakeRandom` class to generate a population of neurons of a certain type with random attributes.
- The `make_default` function to generate a population of neurons of a certain type with the default attributes.
- The `make_clone` function to generate neurons of a certain type whose attributes are copied from those of a given neuron.

projection namespace

The `projection` namespace contains the `creators`, `synapse_generators`, and `parameters_generators` namespaces.

The namespace also implements the `connect_populations` function to link two projections. As a result, the function creates a projection connecting these two populations and adds it to the neural network object. Use this function to link two populations with one of the following images:

- The resulting projection has arbitrary connections between population neurons.
- In the resulting projection, each neuron in the presynaptic population is connected to each neuron in the postsynaptic population.

creators namespace

The `creators` namespace implements functions for generating synapse projections. The namespace provides the following functions:

- `all_to_all`—generates a projection of synapses connecting each neuron in the presynaptic population to each neuron in the postsynaptic population.
- `one_to_one`—generates a projection of synapses connecting neurons of the presynaptic and postsynaptic populations in a one-to-one relationship.
- `from_container`—generates a projection of synapses from a container containing synapse attributes and presynaptic and postsynaptic neuron indices.
- `from_map`—generates a projection of synapses from a `std::map` object containing synapse attributes as value and presynaptic and postsynaptic neuron indices as key.
- `fixed_probability`—generates a projection of the synapses connecting each neuron in the presynaptic population to each neuron in the postsynaptic population. Synapses are generated with a given probability.
- `index_based`—generates a projection of synapses connecting neurons in the population if the synapse generation function returns a non-zero `std::optional` value. The indices of the presynaptic and postsynaptic neurons are passed to the synapse generation function.
- `fixed_number_post`—generates a projection of synapses connecting each neuron in the presynaptic population to a given number of random neurons in the postsynaptic population.
- `fixed_number_pre`—generates a projection of synapses connecting each neuron in the postsynaptic population to a given number of random neurons in the presynaptic population.
- `clone_projection`—generates a projection whose synapses are copied from another projection. The original and created projections can be of different types.

synapse_generators namespace

The `synapse_generators` namespace implements functions and interfaces for generating synapses. The namespace provides the following functions and interfaces:

- The `all_to_all` function, which generates synapses between each neuron in the presynaptic population and each neuron in the postsynaptic population.
- The `one_to_one` function, which generates one-to-one synapses between neurons in the presynaptic and postsynaptic populations.
- The `from_container` function, which generates synapses from a container containing synapse attributes and presynaptic and postsynaptic neuron indices.
- The `FromMap` class, which generates synapses from an `std::map` or `std::unordered_map` object containing synapse attributes as the value and presynaptic and postsynaptic neuron indices as the key.
- The `FixedProbability` class, which generates synapses between each neuron in the presynaptic population and each neuron in the postsynaptic population. Synapses are generated with a given probability.
- The `index_based` function, which generates synapses connecting neurons in the populations if the synapse generation function returns a non-zero `std::optional` value. The indices of the presynaptic and postsynaptic neurons are passed to the synapse generation function.
- The `FixedNumberPost` class, which generates synapses between each neuron in the presynaptic population and a specified number of random neurons in the postsynaptic population.

- The `FixedNumberPre` class, which generates synapses between each neuron in the postsynaptic population and a specified number of random neurons in the presynaptic population.
- The `Clone_projection` function, which generates synapses whose attributes are copied from another synapse. The original and newly created synapses can have different types.

parameters_generators namespaces

The namespace implements the `default_synapse_gen` function and the `CopySynapseGen` class. The `default_synapse_gen` function generates a synapse with the default attribute values. The `CopySynapseGen` class implements a synapse generator that copies attributes from a given synapse.

input namespace

The `input` namespace implements interfaces for accessing the input channel and the converter for data coming from the external environment. Data can be passed to Kaspersky Neuromorphic Platform synchronously (upon the user request) and asynchronously (upon the call of the handler).

SequenceConverter class

The `SequenceConverter` class implements a data converter that takes the data from the `istream` data stream and divides the incoming data into chunks of a certain length. The data converter uses functions of the `input` namespace to convert the values in the input segments to the `boolean` type. If the input value contains a spike, the function returns `true` value. An object of the `SequenceConverter` class returns a list of indices of the input values that contain spikes as a message of the `SpikeMessage` type.

Each object of the `SequenceConverter` class has the following attributes:

- The `istream` data stream from which the data is received.
- The function for converting input values to the `boolean` type.
- The length of the input data segment.

IndexConverter class

The `IndexConverter` class implements a data converter that takes data from the `istream` data stream and converts a sequence of indexes of the incoming values containing spikes to a message of the `SpikeMessage` type.

Each object of the `IndexConverter` class has the following attributes:

- The `istream` data stream from which the data is received.
- The character used to separate the indexes of values in the message of the `SpikeMessage` type.

InputChannel class

The `InputChannel` class implements an input channel that receives messages of the `SpikeMessage` type from the data converter. Using the functions of the `InputChannel` class, the converted data can be sent through the message endpoints to the subscribers having a subscription.

Each object of the `InputChannel` class has the following attributes:

- Basic input channel data.
- The message endpoint to a message bus through which the converted data is sent to subscribers.
- Converter of data to the messages of the `SpikeMessage` type.

output namespace

The `output` namespace implements an interface for accessing the output channel and `SpikeMessage` message converters. Kaspersky Neuromorphic Platform can transmit data both synchronously (upon the user request) and asynchronously (upon the call of the handler).

OutputChannel class

The `OutputChannel` class implements an output channel that receives a set of messages containing spikes from an associated population over several neural network execution cycles. The messages of the `SpikeMessage` type can be converted to one of the following data formats, defined by message translators:

- A vector of `boolean` values.
- A vector of values, where each value corresponds to the number of spikes generated by the neuron.
- A vector of neurons that generated spikes from a sequence of messages of the `SpikeMessage` type.

If these data formats are not suitable for you, you can implement your own function to convert the received messages to the desired data format.

ConvertToSet class

The `ConvertToSet` class implements a data converter that receives messages of the `SpikeMessage` type from an object of the `OutputChannel` class and converts them to a vector of neurons that generate spikes.

Each object of the `ConvertToSet` class contains the size of the neuron vector.

converter_bitwise functor

The `converter_bitwise` functor implements a data converter that receives the messages of the `SpikeMessage` type and converts them to a vector of `boolean` values, where `true` indicates a spike.

converter_count functor

The `converter_count` functor implements a data converter that receives the messages of the `SpikeMessage` type and converts them to a vector of values. Each value of the vector corresponds to the number of spikes generated by the neuron.

monitoring namespace

The `monitoring` namespace contains the `MessageObserver` class. The `MessageObserver` class implements an observer that subscribes to a particular set of entities and a particular type of messages. At the end of each neural network execution cycle, the observer receives messages from the entities by subscription and processes the messages in accordance with the specified function.

Each object of the `MessageObserver` class has the following attributes:

- Message endpoint of the message bus that provides a subscription for the message from the entities.
- Function for processing the received messages.
- Observer ID.

Network class

The `Network` class implements a neural network object. A neural network object stores basic data, populations, projections, and strategies for generating neurons and synapses, as well as provides a common interface for accessing neurons and synapses of the neural network. In a neural network, all populations are connected by projections.

The class contains functions that you can use to add or remove neural network populations and projections, and link these.

Model class

The `Model` class implements a model that stores and connects a neural network and the input and output channels. To execute a model on different backends, you can download the model object into an object of the `ModelExecutor` class and call the `start()` model execution function.

Using the functions of the `Model` class, you can manage the neural network object. For example, you can add to the neural network object input and output channels associated with certain projections.

Each object of the `Model` class has the following attributes:

- Basic model data.
- Neural network.
- Input and output channel IDs.

BackendLoader class

The `BackendLoader` class implements a dynamic library loader that returns a pointer to the loaded backend instance. The pointer to the backend instance is passed to the `ModelExecutor` to execute the model in the backend.

Using the functions of the `BackendLoader` class, you can download dynamic libraries and check whether the downloaded library is a backend library.

The `BackendLoader` class is implemented using the Boost:DLL library.

ModelLoader class

The `ModelLoader` class implements a model loader that uploads the model to the appropriate backend instance. You can use the `ModelLoader` functions to upload the model to the backend and to get information about the input and output channels used in the model.

Each object of the `ModelLoader` class has the following attributes:

- Basic data of the model loader.
- Backend instance where the model is loaded.
- An input channel map containing the ID, the hashed ID, and the input data converter.
- Input and output channels.

ModelExecutor class

The `ModelExecutor` class implements a model executor. Using the `ModelExecutor` class, you can execute the loaded model on different backends. To execute the model, the `ModelExecutor` class uses the `ModelLoader` and a pointer to a backend instance received from a `BackendLoader` class object.

Each object of the `ModelExecutor` class has the following attributes:

- Basic data of the model executor.
- Model loader.
- Observers.

Solving common tasks in C++

This section contains a description of common user tasks that can be solved using C++.

Additionally, you can find examples of using the platform in `/usr/src/knp/examples` directory after installing `knp-examples` package or in the platform repository. The directory with each example contains a `README.md` file with a description of the example in it.

List of examples for C++:

- `mnist-client` is an example of classifying images of handwritten numbers from the MNIST database.
- `Save-load-networks` is the example of saving and loading a neural network in the `SONATA` format.
- `simple-network` is the example of creating and executing a neural network uploaded to the backend automatically.

Adding a new neuron type

Currently, the Kaspersky Neuromorphic Platform supports the property sets and population typification based on the neuron models `AltAILIF`, `BLIFATNeuron`, and `SynapticResourceSTDPBLIFATNeuron`. If required, you can add a new neuron type.

> You can use this instruction when working with the platform source code.

To add a new neuron type:

1. Go from the working directory where the platform source code archive was unpacked to `neuron-trait-library/include/knp/neuron-traits/`, and create a header file for the new neuron type there.
2. In the created header file, define the neuron type structure and its template properties (for example, `default_values`, `neuron_parameters`).

   An example of a BLIFAT neuron definition:

   neuron-trait-library/include/knp/neuron-traits/blifat.h

```
#include "type_traits.h"

namespace knp::neuron_traits

{
struct BLIFATNeuron;

template <>
struct default_values<BLIFATNeuron>

{
 // Specifies the number of steps in the neural network since the last spike step by default.
 constexpr static std::size_t n_time_steps_since_last_firing_ = std::numeric_limits<std::size_t>::infinity();
 // Specifies the default value to which the membrane potential tends for
 // conduction-based inhibitory synapses.
 constexpr static double reverse_inhibitory_potential = -0.3;
 // Specifies the default value to which the membrane potential tends for
 // current-based inhibitory synapses.
 constexpr static double min_potential = -1.0e9;
 // ...
};

template <>
```

```
struct neuron_parameters<BLIFATNeuron>

{
 // Specifies the number of steps in the neural network since the last spike step.
 std::size_t n_time_steps_since_last_firing_ = default_values<BLIFATNeuron>::n_time_steps_since_last_firing_;
 // Specifies the threshold value of the membrane potential of the neuron.
 double activation_threshold_ = default_values<BLIFATNeuron>::activation_threshold_;
 // Specifies a dynamic threshold value of the membrane potential for the neuron to reach
 // before it generates a spike.
 double dynamic_threshold_ = default_values<BLIFATNeuron>::dynamic_threshold_;
 // ...
};

} // namespace knp::neuron_traits
```

3. Add the created neuron type to the list of neurons defined in the `neuron-trait-library/include/knp/neuron-traits/all_traits.h` header file.

   An example of adding a BLIFAT neuron to the list of neurons:

   neuron-trait-library/include/knp/neuron-traits/all_traits.h

```
#include "blifat.h"

namespace knp::neuron_traits

{
// A comma-separated list of neuron types.
#define ALL_NEURONS BLIFATNeuron
// ...
} // namespace knp::neuron_traits
```

4. Implement the overloading startup method in the desired backends.

   For example, for the CPU backend, implement the overloading method for starting the calculation of populations.

Adding a new synapse type

Currently, the Kaspersky Neuromorphic Platform supports the property sets and projection typification based on the synapse models `DeltaSynapse`, `AdditiveSTDPDeltaSynapse`, and `SynapticResourceSTDPDeltaSynapse`. If required, you can add a new synapse type.

> You can use this instruction when working with the platform source code.

To add a new synapse type:

1. Go from the working directory where the platform source code archive was unpacked to `synapse-traits-library/include/knp/synapse-traits/`, and create a header file for the new synapse type.

2. In the created header file, define the synapse type structure and its template properties (for example, `default_values`, `synapse_parameters`).

   An example of a delta synapse definition:

   synapse-traits-library/include/knp/synapse-traits/delta.h

```
#include "type_traits.h"

namespace knp::synapse_traits

{
```

```
struct DeltaSynapse;


template <>
struct default_values<DeltaSynapse>

{
 // Specifies the default synapse weight.
 constexpr static float weight_ = 0.0F;
 // Specifies the default synapse delay.
 constexpr static uint32_t delay_ = 1;
 // Specifies the default synapse type.
 constexpr static OutputType output_type_ = OutputType::EXCITATORY;
};


template <>
struct synapse_parameters<DeltaSynapse>

{
 // Specifies the synapse attributes.
 synapse_parameters() : weight_(0.0F), delay_(1), output_type_(knp::synapse_traits::OutputType::EXCITATORY) {}

 synapse_parameters(float weight, uint32_t delay, knp::synapse_traits::OutputType type)
 : weight_(weight), delay_(delay), output_type_(type)

 // Synapse weight.
 float weight_;
 // Synapse delay.
 std::size_t delay_;
 // Output synapse type.
 knp::synapse_traits::OutputType output_type_;
};

} // namespace knp::synapse_traits
```

3. Add the created synapse type to the list of synapses defined in the `synapse-traits-library/include/knp/synapse-traits/all_traits.h` header file.

   An example of adding a delta synapse to the list of synapses:

   synapse-traits-library/include/knp/synapse-traits/all_traits.h

```
#include "delta.h"

namespace knp::synapse_traits

{
// A comma-separated list of synapse types.
#define ALL_SYNAPSES DeltaSynapse

// ...
} // namespace knp::neuron_traits
```

4. Implement the overloading startup method in the desired backends.

   For example, for the CPU backend, implement the overloading method for starting the calculation of projections.

Execution of a neural network created from projections and populations

This section contains instructions on how to create a neural network from projections and populations, load projections and populations to the CPU backend, and execute the neural network.

> This variant for creating and executing a neural network is used rarely.
> You can use this instruction in the development of application solutions.

To manually create and execute a neural network:

1. In the directory of your project, create a `CPP` application file where the function for creating and running a neural network is implemented.

2. In the application file, include the header files required for the neural network execution using the `#include` directive.

   If necessary, define aliases by means of the `using` statement.

   An example of connecting header files for executing a neural network with a population of BLIFAT neurons and a projection of delta synapse on a single-thread CPU backend:

   ```C++
   #include <knp/backends/cpu-single-threaded/backend.h>
   #include <knp/core/population.h>
   #include <knp/core/projection.h>
   #include <knp/neuron-traits/blifat.h>
   #include <knp/synapse-traits/delta.h>

   #include <vector>


   using Backend = knp::backends::single_threaded_cpu::SingleThreadedCPUBackend;
   using DeltaProjection = knp::core::Projection<knp::synapse_traits::DeltaSynapse>;
   using BLIFATPopulation = knp::core::Population<knp::neuron_traits::BLIFATNeuron>;
   using Population = knp::backends::single_threaded_cpu::SingleThreadedCPUBackend::PopulationVariants;
   using Projection = knp::backends::single_threaded_cpu::SingleThreadedCPUBackend::ProjectionVariants;
   ```

3. Implement a neuron generator with the desired properties or use an existing generator from the framework library.

   An example of a BLIFAT neuron generator implementation:

   ```C++
   // ...
   // A function is implemented that generates neurons.
   // The function is stored in the neuron_generator variable.
   auto neuron_generator = [](size_t index)
   {
    return knp::neuron_traits::neuron_parameters<knp::neuron_traits::BLIFATNeuron>{};
   };
   // ...
   ```

4. Implement a synapse generator with the desired properties or use an existing generator from the framework library.

   An example of a delta synapse generator implementation:

   ```C++
   // ...
   // A function is implemented that generates synapses for the projection connected
   // to an input channel. The function is stored in the input_projection_gen variable.
   DeltaProjection::SynapseGenerator input_projection_gen = [](size_t index) ->
   std::optional<DeltaProjection::Synapse>
   {
    return DeltaProjection::Synapse{{1.0, 1, knp::synapse_traits::OutputType::EXCITATORY}, 0, 0};
   };

   // A function is implemented that generates synapses for the projection which loops
   // the output of the population to itself. The function is stored in the synapse_generator variable.
   DeltaProjection::SynapseGenerator synapse_generator = [](size_t index) ->
   std::optional<DeltaProjection::Synapse>
   {
    return DeltaProjection::Synapse{{1.0, 6, knp::synapse_traits::OutputType::EXCITATORY}, 0, 0};
   };
   // ...
   ```

5. Create a population object and pass the created neuron generator to the constructor.

   An example of a population object creation:

```C++
// ...
// Creates a population object.
BLIFATPopulation population{neuron_generator, 1};
```

6. Create a projection object that loops the output of the population to itself. Pass the associated population ID and the synapse generator to the constructor.

   An example of creating a projection that loops the output of a population to itself:

```C++
// ...
// Creates an object of the projection that loops the output of the population to itself.
Projection loop_projection = DeltaProjection{
 population.get_uid(), population.get_uid(), synapse_generator, 1
};
// ...
```

7. Create an input projection object. Pass the synapse generator and associated population ID to the constructor.

   An example of creating an input projection connected to the input channel and population:

```C++
// ...
// Creates an object of the input projection, which is assigned a zero ID
// (knp::core::UID{false}). The projection receives spikes from the input channel and sends
// synaptic impact to the population object.
Projection input_projection = DeltaProjection{
 knp::core::UID{false}, population.get_uid(), input_projection_gen, 1
};

// Gets the UID of the input projection that sends the synaptic impact to the population.
knp::core::UID input_uid = std::visit([](const auto &proj) { return proj.get_uid(); }, input_projection);
// ...
```

8. Upload the population and projections to the backend.

   An example of uploading a population and projections to the backend:

```C++
// ...
Backend backend;
// ...
// Uploads the created population object to the backend.
backend.load_populations({population});

// Uploads the created projection objects to the backend.
backend.load_projections({input_projection, loop_projection});
```

9. Define a message bus endpoint object.

   Example of creating a message bus endpoint:

```C++
// ...
// Create message bus endpoint.
auto endpoint = backend.message_bus_.create_endpoint();
// ...
```

10. Connect an input channel to the input projection; connect the population to an output channel. Define unique IDs for the input and output channels and generate messages for them.

    An example of connecting to the input and output channels:

```C++
// ...
// Creates an arbitrary UID of the input channel.
knp::core::UID in_channel_uid;

// Creates an arbitrary UID of the output channel.
knp::core::UID out_channel_uid;

// Creates a spike subscription from the input channel.
backend.subscribe<knp::core::messaging::SpikeMessage>(input_uid, {in_channel_uid});

// Creates a spike subscription from the population with the specified UID for the output channel
// with the specified UID.
endpoint.subscribe<knp::core::messaging::SpikeMessage>(out_channel_uid, {population.get_uid()});
...
```

11. Configure sending of messages in the neural network and specify the parameters of the neural network execution cycles.

    The following example of a neural network execution consists of 20 steps:

```C++
// ...
std::vector<size_t> results;
for (size_t step = 0; step < 20; ++step)
{
 // Sends the input projection spikes from the input channel at the following steps: 0, 5, 10, and 15.
 if (step % 5 == 0)
 {
 knp::core::messaging::SpikeMessage message{{in_channel_uid, 0}, {0}};
 endpoint.send_message(message);
 }
 backend.step();
 endpoint.receive_all_messages();
 // Receives spikes sent by the population to the output channel.
 auto output = endpoint.unload_messages<knp::core::messaging::SpikeMessage>(out_channel_uid);

 // Writes the indices of the steps at which spikes are received by the output channel.
 if (!output.empty()) results.push_back(step);
}
// ...
```

[Neural network execution steps]

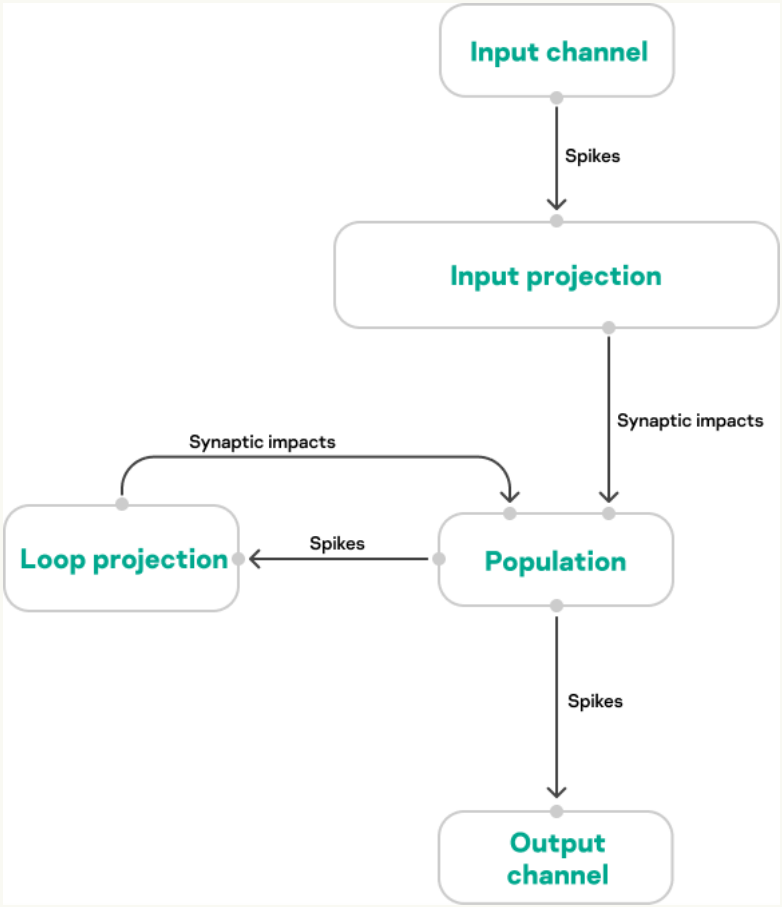
The neural network is executed in loops. The actions described in the loop are repeated every 5 steps of the network execution.

At the first step of the loop, the input channel sends the input projection spikes.

At the next step of the loop, the input projection computes synaptic impacts and sends them to the populations. After receiving a synaptic impact, the population sends spikes to the output channel and to the projection that loops the output of the population to itself.

After 6 steps of the loop are executed since the spikes sent by the input projection are received, the projection that loops the population output to itself calculates the synaptic impacts and sends the impact to the population.

The figure below shows a diagram of the neural network execution.



Neural network execution diagram

In this example, the neural network is executed in a loop with the following steps:

- *Step 0*. The input channel sends the input projection spikes. This step is repeated every 5 steps of the neural network execution loop.

- *Step 1*. The input projection computes synaptic impacts and sends them to the populations. After receiving a synaptic impact, the population sends spikes to the output channel and to the projection that loops the output of the population to itself. This step is repeated every 5 steps of the neural network execution.

- *Step 7*. The projection that loops the population output to itself computes the synaptic impacts after receiving the spikes sent 6 steps before and sends the impacts to the population. The population sends spikes to the output channel and to the projection that loops the output of the population to itself. This step is repeated every 5 steps of the neural network execution.

- *Step 13*. The projection that loops the population output to itself computes the synaptic impacts after receiving the spikes sent 6 steps before and sends the impacts to the population. The population sends spikes to the output channel and to the projection that loops the output of the population to itself. This step is repeated every 5 steps of the neural network execution.

- *Step 19*. The projection looping back the population inference calculates the synaptic influences after receiving the spikes sent in step 13, and transmits the population influences. The population sends spikes to the output channel and to the projection that loops the output of the population to itself.

12. Save the file.

If required, build the application solution using CMake.

Python platform components

This section describes the Kaspersky Neuromorphic Platform components written in Python.

neuron_traits module

The `neuron_traits` module is a set of neuron attributes. You can use the neuron attributes provided in the module, for example, when constructing populations.

Currently, the module supports the property sets for the following neurons:

- Neuron based on the `BLIFATNeuron` model (hereinafter also BLIFAT neuron).

  A neuron based on the `BLIFATNeuron` model is an integrator neuron with a leak and an adaptive threshold, which generates a series of spikes (Bursting Leaky Integrate-and-Fire with Adaptive Threshold Neuron). A BLIFAT neuron is characterized by the membrane potential and the threshold value of the membrane potential.

  The value of the membrane potential changes in accordance with the value of the synaptic impact coming from the associated synapse. If the neuron membrane potential reaches a threshold value, the neuron starts generating a series of spikes with a certain interval. In this case, after the spike generation, the membrane potential tends to the base value, and the threshold value changes in accordance with the increment.

  If a neuron does not receive messages from a connected synapse for a certain time, the membrane potential and its threshold value exponentially tend to the base values.

  You can specify the base value of the membrane potential, the base threshold value, the increment value, the time interval upon reaching which the values of the membrane potential and potential threshold begin to approach the base value, the duration of a spike series, and the interval for spike generation in the attributes of a neuron instance.

- Neuron based on the `SynapticResourceSTDPBLIFATNeuron` model.

  A neuron based on the `SynapticResource STDPBLIFATNeuron` model is a BLIFAT neuron characterized by a free synaptic resource, a threshold value of a free synaptic resource, and stability.

  If the absolute value of the free synaptic resource exceeds the threshold value, this free synaptic resource is distributed equally among all plastic presynaptic connections of the neuron. After that, the value of the free synaptic resource of the neuron equals `0`. If the absolute value of the free synaptic resource is less than the threshold value, the neuron appropriates the synaptic resource from the connected synapses until the threshold value is reached.

  The neuron stability value changes in accordance with the synaptic plasticity used by the synapse and affects its plasticity value.

If necessary, you can implement and use your own neuron models when working with the source code.

synapse_traits module

The `synapse traits` module is a set of synapse attributes. You can use the synapses provided in the module, for example, when creating projections.

Currently, the module supports the property sets for the following synapses:

- Synapse based on the `DeltaSynapse` model (hereinafter also referred to as delta synapse).

After receiving spikes from a presynaptic population, the synapses generate a message containing the synaptic impact values. The synapse projection sends the synaptic impact to the population to change the membrane potential of the postsynaptic neuron. In this case, the message is sent by the projection without a time delay.

The sent synaptic impact is determined by the weight of the delta synapse. You can specify the synapse weight in the attributes of its instance.

- Synapse based on the `SynapticResourceSTDPDeltaSynapse` model.

  A synapse based on the `SynapticResourceSTDPDeltaSynapse` model is a delta synapse whose synaptic plasticity depends on the synaptic resource. The value of the synaptic resource depends on the synaptic plasticity:

  - Hebbian plasticity.

    The value of a synaptic resource depends on the period of the dense sequence of spikes (hereinafter also referred to as DSS). *DSS* is a sequence of unforced spikes generated by a neuron, where all adjacent spikes are separated by the time intervals by no more than *ISImax*, where *ISImax* is the maximum allowed distance between spikes in the DSS. The *DSS period* is the time interval between the moment when the Hebbian plasticity starts before the first spike in the DSS and one of the following moments: when the maximum allowable distance ISImax is reached after the last spike in the DSS, when a forced activation occurs, or when the spike arrives at the dopamine synapse.

    When at least one spike is received within the DSS period, the synaptic resource changes by the value of the current Hebbian plasticity. The Hebbian plasticity depends on the neuron stability variable and is calculated using the formula shown below, where:

    - d_H corresponds to Hebbian plasticity.
    - s corresponds to the neuron stability variable. At the time of the first spike within the DSS period, the stability value changes by a constant. If the stability is negative, it does not change downward.

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

    Formula for calculating Hebbian plasticity

  - Unconditional plasticity.

    In the case of a nonzero unconditioned plasticity, the value of the synaptic resource decreases by the value of plasticity when a spike is received. The unconditional plasticity depends on the neuron stability variable and is calculated using the formula shown below, where:

    - d_H corresponds to unconditional plasticity.
    - *s* corresponds to the neuron stability variable. At the time of the first spike within the DSS period, the stability value changes by a constant. If the stability is negative, it does not change downward.

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

    Formula for calculating unconditional plasticity

  - Dopamine plasticity.

    The dopamine plasticity is applicable when the sum of all dopamine synapses that receive a spike at the current step of the neural network execution is not zero. In the case of dopamine plasticity, the value of the synaptic resource changes based on the formula below, where:

    - *D* corresponds to the sum of dopamine synapses.
    - *s* corresponds to the neuron stability variable.

$$D\min(2^{-s}, 1)$$

Formula for calculating synaptic resource in case of dopamine plasticity

The sum of dopamine synapses affects the neuron stability variable. If at the time of the forced activation, the *dopamine punishment* is received (the sum of dopamine synapses is less than zero), the value of the neuron stability is decreased by the *rD* value, where *r* is a constant and *D* is dopamine punishment.

If at the time of the forced activation, the *dopamine reward* is received (the sum of dopamine synapses is greater than zero), the value of the neuron stability is increased by the *rD* value, where *r* is a constant and *D* is dopamine reward. Otherwise, the stability value changes to the value shown below, where:

- *D* corresponds to the sum of dopamine synapses.
- *ISImax* corresponds to the maximum acceptable distance between spikes in the DSS.
- t_{TSS} corresponds to the time since the first spike in the last DSS.

$$D\max\left(2 - \frac{|t_{TSS} - ISI_{max}|}{ISI_{max}}, -1\right)$$

Formula for calculating the neuron stability change in case of the dopamine reward

The synapse weight determines the value of the synaptic impact sent to the postsynaptic population and is associated with the value of the synaptic resource by the formula shown below, where:

- *w* corresponds to the synapse weight.
- *W* corresponds to a synaptic resource.
- $\boldsymbol{w}_{\min}$ corresponds to a random minimum weight.
- $\boldsymbol{w}_{max}$ corresponds to a random maximum weight.

$$\boldsymbol{w} = \boldsymbol{w}_{\min} + \frac{(\boldsymbol{w}_{max} - \boldsymbol{w}_{min}) * max(W, 0)}{\boldsymbol{w}_{max} - \boldsymbol{w}_{min} + max(W, 0)}$$

Formula for calculating the weight change depending on the synaptic resource

If needed, you can implement and use your own synapse models when working with source code.

core module

The `core` module provides support for backends. The `core` module provides the classes required for interacting with backends and performing application tasks.

The table below contains descriptions of the module components.

The core module components

Component	Description
`Backend`	Base class for specific backend implementations.
BackendWrapper	Structure that implements a wrapper for the Backend base class.
`BaseData`	Structure defining a set of base data.
`continuously_uid_generator`	Class implementing a generator of unique IDs.
`MessageBus`	Class implementing an interface for the message bus.

`MessageEndpoint`	Class implementing an interface for the message bus endpoint.
`Population`	Set of classes that implement neuron containers of the same model.
`Projection`	Set of classes that implement synapse containers of the same type.
`Subscription`	Set of classes that implement a message subscription.
`TagMap`	Class implementing an interface for the tag dictionary.
`UID`	Class defining unique identifiers.
`uid_hash`	Class that implements hashing of UIDs.
`UUID`	Class that implements the interface to the UID of the Boost:Uuid library.
`messaging`	Namespace that contains message interfaces.

The figure below shows how the module components interact.

In the Kaspersky Neuromorphic Platform, the `BaseData` structure contains basic information about an object and includes `UID` and `TagMap`. The `BaseData` structure is used in the following classes: `Projection`, `Population`, and `Backend`. Therefore, each of them has a `UID` and a set of tags with the `TagMap` values.

The `Population` and `Projection` classes receive and send messages that are implemented in the classes of the `messaging` namespace. Messages contain the sender's UID in the header. Message exchange is implemented using the functions of the `Backend` class, which stores the `MessageBus` and `MessageEndpoint` classes. The `MessageBus` creates `MessageEndpoint` and interacts with them by exchanging messages. Each `MessageEndpoint` has a container of the `Subscription` class.

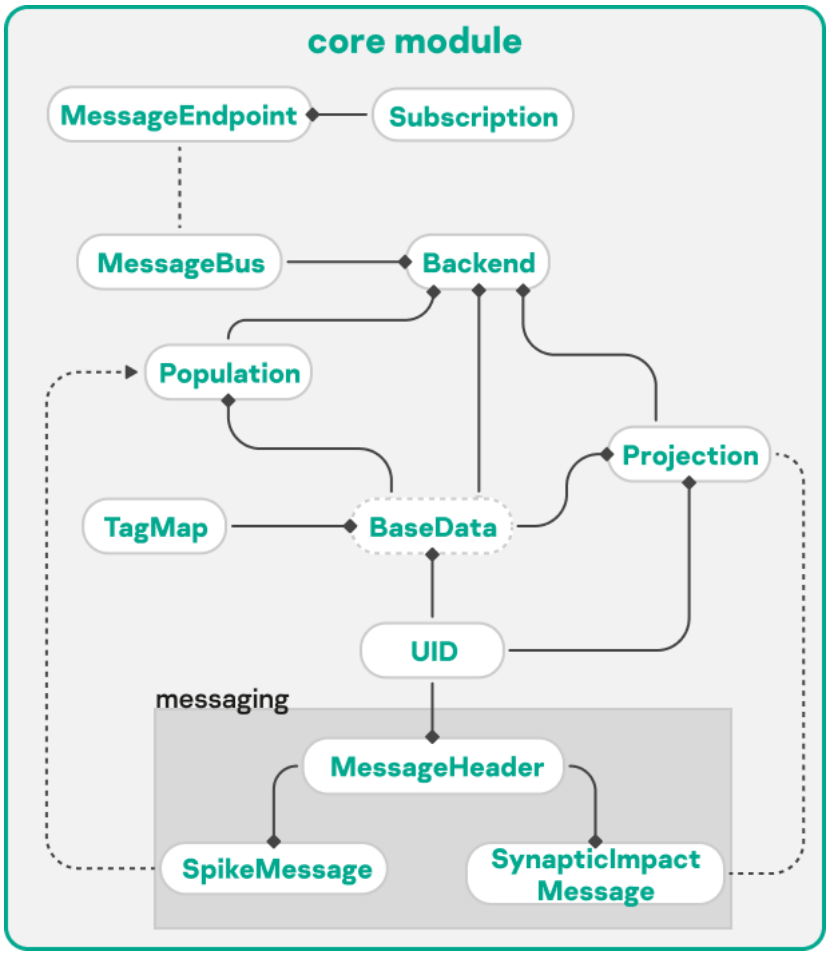


Diagram of interactions between components in the core module

UID class

An object of the `UID` class is part of the `BaseData` structure; it represents the unique identifier of the object.

If necessary, you can generate a random UID or create a unique `UUID`. The `UID` class also provides logical, comparison, and assignment operators.

uid_hash class

The `uid_hash` class implements the object ID hashing function.

UUID class

The `UUID` class implements the interface to the UID of the `Boost:Uuid` library. Each unique ID has the following properties:

- Size of the unique ID in bytes
- Function that returns `true` if the unique ID consists solely of zeros.
- One of the following ID options:
  - `NCS`—reserved for backward compatibility with the Apollo Network Computing System 1.5 UUID format.
  - `RFC_4122`—described in RFC 4122.
  - `MICROSOFT`—reserved for backward compatibility with early Microsoft Windows GUIDs.
  - `FUTURE`—reserved for future use.

continuously_uid_generator class

The `continuously_uid_generator` class implements a unique ID generator (UID). The ID is displayed as a string divided into groups by means of dashes and is a 128-bit number. IDs are generated sequentially in increments of 1.

You can use the `continuously_uid_generator` class to debug your platform and execute the neural network.

TagMap class

An object of the `TagMap` class is a part of the `BaseData` structure. It is an associative array that includes zero or more "tag key-tag value" records, where the tag name is the key. Tag names are unique strings.

BaseData structure

The `BaseData` structure defines a set of base data common to multiple `core` module objects and framework objects. For example, `BaseData` is used in objects of the `Backend`, `Population`, and `Projection` classes.

Base data `BaseData` has the following properties:

- Unique identifier of the object
- Associative `TagMap` array consisting of the tags used by the object and their values

Set of Population classes

The set of `Population` classes implements containers, each of which contains neurons of the same model. When you use the set of `Population` classes, you can specify one of the neuron types supported by the `neuron_traits` module. Each population of neurons has a unique ID.

Using the set of `Population` classes, you can manage neurons within a population. You can also upload populations to the backend to compute the neural network. The obtained population properties are stored on the backend. To pass new properties from the backend to the corresponding population objects, call the synchronization. If you manually change the population properties before calling synchronization, you can upload the population to the backend again to compute a new neural network.

Set of Projection classes

The set of `Projection` classes implements containers; each container contains synapses of one of the types supported by the `synapse_traits` module. Each synapse projection has a unique ID.

Projection classes store synapses, which are connections between the neurons in the presynaptic and postsynaptic populations. The IDs of these populations are stored in projection.

Using the set of `Projection` classes, you can manage synapses within a projection, as well as send connections between the neurons to the backend and receive them from the backend.

Backend class

The `Backend` class is the base class for backend implementations. Each backend is implemented as a library, allowing you to:

- Run training and execution of the spiking neural network on various devices
- Receive messages containing spikes and synaptic impacts for a given message endpoint of the message bus
- Get internal data to monitor the neural network operation
- Upload internal data of the neural network operation monitoring received from the computer

To initialize the backend, first download projections containing synapses and populations containing neurons. During initialization of the backend, the loaded synapse projections are subscribed to the spikes of the associated populations or input channels, and the downloaded populations are subscribed to synaptic impacts of the associated projections.

Each object of the `Backend` class has the following properties:

- Unique ID of the backend.
- Message bus. Using the message bus, the backend ensures the message exchange between the objects of the `core` module, as well as sending messages from modifiers to neurons and synapses.
- Message bus endpoint.
- Neural network execution status on the backend.

BackendWrapper structure

The `BackendWrapper` structure implements a wrapper for the `Backend` base class to use its objects in Python.

Use `BackendWrapper` to manage synaptic plasticity support for a `Backend` object, and get information about neurons, synapses, populations and projections supported by the backend instance, and devices and endpoints of the message bus used. You can load neuron synapse and population projections into the backend instance and delete them to execute the neural network, and manage training.

MessageBus class

The `MessageBus` class implements an interface for accessing the message bus for receiving, distributing, and delivering messages between objects connected to it An object of the `MessageBus` class provides many message endpoints of the message bus through which the objects of the `core` module can subscribe to the messages or send them.

The `MessageBus` class provides functions that you can use to create message endpoints and pass messages through the message bus. If you delete a message endpoint, message transmission through this message endpoint stops.

Using a message bus allows you to implement the publisher-subscriber principle. The publisher sends messages through a message bus as a proxy. The subscription object filters messages and sends them to the subscribers. This approach allows you to avoid creating a large number of IPC channels because the publisher and subscribers are not connected directly.

MessageEndpoint class

The `MessageEndpoint` class implements an interface for accessing a message endpoint that subscribes to the messages and sends them using the message bus.

An object of the `MessageEndpoint` class stores the message recipient ID and the subscription class key that is used to filter messages. A message bus can have multiple message endpoints.

Using the functions of the `MessageEndpoint` class, you can configure the exchange of messages between the backend and the framework objects with the specified IDs. To send messages, first add a subscription class key to the message endpoint object.

If you delete a message endpoint, the associated subscription classes are also deleted. Message transmission through this message endpoint stops.

Set of Subscription classes

The set of `Subscription` classes implements the functionality of subscription to messages. A class from the `Subscription` group is used to store messages of a specific type until they are requested by the recipient. You can use the functions of the set of `Subscription` classes to define the recipients of stored messages.

Subscription class keys are stored at the message endpoints of the message bus. If the message endpoint object that stores the class key is deleted, the corresponding subscription class is also deleted. Sending of messages defined by this class is stopped.

messaging namespace

The `messaging` namespace implements interfaces for accessing messages. The message contains data sent and received by the backend objects, as well as data sent by the objects identified by UID. Message exchange is provided by the `MessageBus` message bus.

Each message has the following properties:

- `MessageHeader`—message header containing the following information:
  - `sender_uid`—ID of the object that sends the message.
  - `send_time`—the time when the message was sent.
- The message body, consisting of the set of transmitted attributes. The set of the transmitted properties depends on the message type.

Messages are used to update user structures and synchronize changes on the backend. The table below contains a description of the supported backend message types.

Message structures

Message type	Description	Message structure
`SpikeMessage`	A message from a neuron population to a synapse projection containing a spike. More than one projection can be subscribed to the messages containing spikes.	`header`—message header containing the UID of the population whose neurons generated the spikes and the time when the spikes were generated. `neuron_indexes`—indices of the neurons in the population that generated spikes. Neuron indices are unique within a population.
`SynapticImpactMessage`	A message from a synapse projection to the neuron population containing synaptic impact.	`header`—message header containing the projection UID and the time when the message was sent. `postsynaptic_population_uid` — UID of the postsynaptic population. `Presynaptic_population_uid` — UID of the presynaptic population. `is_forcing` — boolean value indicating the use of plasticity in the synapse projection. If the synapse projection uses synaptic plasticity, the attribute is `true`, otherwise it is `false`. `impacts` — vector containing the following attributes: • `connection_index` — connection index. • `impact_value` — synaptic impact value. • `synapse_type` — type of synapse. • `presynaptic_neuron_index` — index of the presynaptic neuron from which the spike was received by the projection synapse. The neuron index is unique within the population.

		• `postsynaptic_neuron_index` — postsynaptic neuron index. The value of the synaptic impact is sent to this neuron to change the values of its attributes. The neuron index is unique within the population. The `impacts` vector may contain more than one set of the listed attributes to convey synaptic impacts to multiple neurons in the same population.

Python framework

Python framework is designed for creating spiking neural networks and converting artificial neural networks into spiking networks. Resulting spiking neural networks are trained with synaptic plasticity methods and can be executed on a central processor, an AltAI-1 neuromorphic processing unit, or a software emulator of the latter. Artificial neural networks converted into spiking networks with `ANN2SNN` package ternary layers are trained through backpropagation that considers the limitations of the AltAI-1 neuromorphic processing unit. Resulting neural networks are adapted for execution on the AltAI-1 neuromorphic processing unit, or a software emulator of the latter.

Python framework is a set of independent units that manage backends, transform the neural network representation into various formats, and select the optimal neural network to run. Python framework implements functions that do not directly relate to training and execution of spiking neural networks.

Using the Python framework for Kaspersky Neuromorphic Platform, you can dynamically load various backends.

The Python framework is implemented in the `python-framework` module. The table below contains descriptions of the Python framework components.

Python framework components

Component	Description
`ANN2SNN`	A package that implements ternary layers of a neural network.
`input`	A namespace that implements an interface for accessing the input channel.
`output`	A namespace that implements an interface for accessing the output channel.
`sonata`	A namespace that contains functions to load and save the neural network in the `SONATA` format.
`BackendLoader`	A class that implements a dynamic library loader.
`ImpactMessageObserver`	A class that implements the `SynapticImpactMessage` observer.
`Model`	A class that implements the model.
`ModelExecutor`	A class that implements a model executor.
`ModelLoader`	A class that implements the model loader.
`Network`	A class that implements a neural network object.
`SpikeMessageObserver`	A class that implements the `SpikeMessage` observer.

ANN2SNN package

The `ANN2SNN` package is intended for creating neural networks that can be converted into spiking neural networks and placed for execution on the AltAI-1 neuromorphic processor. The `ANN2SNN` package is implemented in Python.

To create neural networks, it is necessary to use special ternary layers, which take into account the limitations typical of the AltAI-1 neuromorphic processor. The ternary layers used to create the neural network correspond to the layers in the Keras library, with the following main differences:

- The input and output data of each of the layers of the neural network are binary tensors of the corresponding dimension.
- All weights of the neural network layers are in the range `{-1,0,1}`.
- The bias value in the neural network layer is in the range from `-32,768` to `32,767`.

When training a neural network created using ternary layers, the following features should be taken into account:

- If non-binarized data exists for training and/or validation, it must first be converted to a binary form. For this, you can use encoding and decoding algorithms or create neural network training blocks that encode and decode data.
- When training binary classification models using the gradient descent method, for the differentiability of the loss function, you must specify the sigmoid function as the output layer activation function. After the training, you will need to replace the sigmoid activation of the output layer with activation using the Heaviside function (Heaviside step function).
- When training multi-class classification models with the gradient descent method, for the differentiability of the loss function, you must specify the multivariate logistic function (softmax function) as the output layer activation function. When such a neural network is placed, the output vector is automatically converted to the binary form that implements the indicator of the maximum output vector component.

Serialization and deserialization of neural networks is performed using standard methods provided by the Keras library, with the `custom_objects` parameter specified in the form of a dictionary of ternary layers and the `Heaviside` activation function.

During placement, the neural network is validated against the processing unit's constraints. The neural network is validated against the neuromorphic processing unit configuration settings specified in the YAML file. As a result of placement, information about the number of cores required to run a neural network on the AltAI-1 neuromorphic processor is displayed. The neural network is converted into a format supported by the processing unit, to be then executed on the hardware processor or its software emulator later. If the neural network cannot be placed on the neuromorphic processing unit because the maximum allowed number of cores is exceeded, an appropriate error is displayed.

> The AltAI-1 neuromorphic processor provides the ability to scale by increasing the number of VLSI chips. For questions regarding scaling, please contact Kaspersky specialists.

You can make a rough estimate of the number of cores required to execute a neural network using the formula below, where:

- *x* corresponds to the number of cores required. After calculating the number of cores using the formula, round the result up.
- *i* corresponds to the ordinal number of the neural network layer.
- *n* corresponds to the number of layers of the neural network.
- *w* corresponds to the first dimension of the tensor of the neural network layer.
- *h* corresponds to the second dimension of the tensor of the neural network layer.

- *c* corresponds to the third dimension of the tensor of the neural network layer.

$$x = \sum_{i=1}^{n} \frac{w_i * h_i * c_i}{512}$$

Formula for estimating the number of cores

The number of cores required for placing a fully connected `TernaryDense` layer depends on the number of input neurons in the layer. If the number of input neurons does not exceed 512, you can calculate the required number of cores by using the formula below, where:

- *x* corresponds to the number of cores required. After calculating the number of cores using the formula, round the result up.
- *i* corresponds to the ordinal number of the neural network layer.
- *n* corresponds to the number of fully connected layers of the neural network.
- *U* is the number of target neurons.

$$x = \sum_{i=1}^{n} \frac{U}{512}$$

Formula for estimating the number of cores for the TernaryDense layer

If the number of input neurons exceeds 512, you can calculate the required number of cores by using the formula below, where:

- *x* corresponds to the number of cores required. After calculating the number of cores using the formula, round the result up.
- *i* corresponds to the ordinal number of the neural network layer.
- *n* corresponds to the number of fully connected layers of the neural network.
- *U* is the number of target neurons.
- *N* is the number of input neurons for the layer.

$$x = \sum_{i=1}^{n} \frac{U}{512} * (\frac{N}{512} + 1)$$

Formula for estimating the number of cores for the TernaryDense layer

For neural networks created on the basis of the `ANN2SNN` package, the following inference options are possible:

- Inference for the number of ticks equal to the number of layers in the neural network. With this inference option, only one layer of the neural network is calculated per tick of neural network execution, and the processed data is transmitted to the next layer. The inference time is directly proportional to the number of layers in the neural network.
- Inference per tick. With this inference option, each layer of the neural network receives data from the previous layer and processes it. The inference time is about 1.5 ms.

Ternary layers of the ANN2SNN package

The following ternary layers are supported in the `ANN2SNN` package:

Classes of the ANN2SNN package that implement ternary layers.

Class	Description
`TernaryDense`	Implements a fully connected layer, which is analogous to the `Dense` layer of the `Keras` library. When working with a fully connected layer, it is recommended to specify the number of neurons in the layer (the `units` parameter) equal to or less than `512`. If a neuron of a layer has more than `512` connections, such a neuron will be represented as several neurons on the AltAI-1 neuromorphic processor.
`TernaryConv1D`	Implements a one-dimensional convolutional layer, which is analogous to the `Conv1D` layer of the `Keras` library. When working with this layer, it is recommended to set the following restrictions: • Kernel size (`kernel_size` parameter)—no more than `20`. • Number of filters (the `filters` parameter)—no more than `64`.
`TernaryConv2D`	Implements a two-dimensional convolutional layer, which is analogous to the `Conv2D` layer in the `Keras` library. When working with this layer, it is recommended to follow the following restrictions: • Do not use large kernels (`kernel_size` parameter). The optimal kernel sizes are `[2, 2]`, `[3, 3]` and `[4, 4]`. • Limit the size of input images to 128x128 and the number of channels to `10`. • Limit the number of filters (`filters` parameter) to `32`.
`TernaryLocallyConnected1D`	Implements a one-dimensional locally connected layer, which is analogous to the `LocallyConnected1D` layer of the `Keras` library. When working with this layer, it is recommended to set the following restrictions: • Kernel size (`kernel_size` parameter)—no more than `20`. • Number of filters (`filters` parameter)—no more than `64`.
`TernaryLocallyConnected2D`	Implements a two-dimensional locally connected layer, which is analogous to the `LocallyConnected2D` layer of the `Keras` library. When working with this layer, it is recommended to follow the following restrictions: • Do not use large kernels (`kernel_size` parameter). The optimal kernel sizes are `[2, 2]`, `[3, 3]` and `[4, 4]`. • Limit the size of input images to 128x128 and the number of channels to `10`. • Limit the number of filters (`filters` parameter) to `32`.
`TernaryDepthwiseConv1D`	Implements a one-dimensional convolutional layer, which is analogous to the `DepthwiseConv1D` layer of the `Keras` library. When working with this layer, it is recommended to set the following restrictions: • Kernel size (`kernel_size` parameter)—no more than `20`. • Number of filters (`filters` parameter)—no more than `64`.
`TernaryDepthwiseConv2D`	Implements a two-dimensional convolutional layer, which is analogous to the `DepthwiseConv2D` layer in the `Keras` library. When working with this layer, it is recommended to follow the following restrictions: • Do not use large kernels (`kernel_size` parameter). The optimal kernel sizes are `[2, 2]`, `[3, 3]` and `[4, 4]`. • Limit the size of input images to 128x128 and the number of channels to `10`. • Limit the number of filters (`filters` parameter) to `32`.
`MaxPool1D`	The standard implementation of the `MaxPool1D` layer of the Keras library `is supported`. To reduce the amount of neuromorphic processor hardware required to execute the neural network, it is recommended to eliminate the use of this layer by increasing the `stride` parameter in the convolutional layers.
`MaxPool2D`	The standard implementation of the `MaxPool2D` layer of the Keras library `is supported`. To reduce the amount of neuromorphic processor hardware required to execute the neural network, it is recommended to eliminate the use of this layer by increasing the `stride` parameter in the convolutional layers.
`BatchNormalization`	The standard implementation of the `BatchNormalization` layer of the `Keras` library is supported. The `scale` parameter is not supported and must be defined as `false`. The `BatchNormalization` layer must come after the `Activation` layer.
`InputLayer`	The standard implementation of the `InputLayer` of the `Keras` library is supported. The input data must be of a binary type.
`Flatten`	The standard implementation of the `Flatten` layer of the `Keras` library is supported.
`Reshape`	The standard implementation of the `Reshape` layer of the `Keras` library is supported.

`Concatenate`	The standard implementation of the `Concatenate` layer of the `Keras` library is supported.

To place the neural network on the AltAI-1 neuromorphic processor, specify the `Heaviside` function as the activation function of each layer. When training a neural network using the gradient descent method, a piecewise-linear approximation of the sigmoid function is used (`hard sigmoid function`).

sonata namespace

The `sonata` namespace implements functions for loading and saving a neural network in the `SONATA` format. The namespace provides the following functions:

- `load_network ()`: load the neural network. The neural network topology must be described in the `SONATA` format.
- `save_network ()`: save a neural network topology in the `SONATA` format.

input namespace

The `input` namespace implements an interface for accessing `InputChannel`. The input channel receives `SpikeMessage` messages from the message generator. `InputChannel` functions can forward received messages to subscribers through endpoints.

output namespace

The `output` namespace implements an interface for accessing `OutputChannel`. The `OutputChannel` class provides functions for reading messages from the message buffer and updating that with messages received from message endpoints.

BackendLoader class

The `BackendLoader` class implements a dynamic library loader that returns a pointer to the loaded backend instance.

Using the functions of the `BackendLoader` class, you can download dynamic libraries and check whether the downloaded library is a backend library.

Network class

The `Network` class implements a neural network object. A neural network object stores populations, projections, and strategies for generating neurons and synapses. It also offers a unified interface for accessing individual neurons and synapses within the network. In a neural network, all populations are connected by projections.

The `Network` class provides functions that you can use to add or delete populations and projections within a neural network.

Each object of the `Network` class has the following properties:

- The number of populations in the neural network.
- The number of projections in the neural network.

Model class

The `Model` class implements a model that stores a neural network and input/output channels, and links these. To execute a model on different backends, you can download the model object into an object of the `ModelExecutor` class and call the `start()` model execution function.

SpikeMessageObserver class

The `SpikeMessageObserver` class implements an observer that subscribes to a particular set of entities and `SpikeMessages`. At the end of each neural network execution cycle, the observer receives messages from the entities by subscription and processes the messages in accordance with the specified function.

ImpactMessageObserver Class

The `ImpactMessageObserver` class implements an observer that subscribes to a particular set of entities and `SynapticImpactMessages`. At the end of each neural network execution cycle, the observer receives messages from the entities by subscription and processes the messages in accordance with the specified function.

ModelLoader class

The `ModelLoader` class implements a model loader that uploads the model to the appropriate backend instance. You can use the `ModelLoader` functions to upload the model to the backend and to get information about the input and output channels used in the model.

ModelExecutor class

The `ModelExecutor` class implements a model executor. Using the `ModelExecutor` class, you can execute the loaded model on different backends. To execute a model, the `ModelExecutor` class uses a pointer to a backend instance obtained from a `BackendLoader` class object.

Solving common tasks in Python

This section contains a description of common user tasks that can be solved using Python.

The Python-based example named mnist-client contains an example for classification of images of handwritten numbers from the MNIST database.

Execution of a neural network manually uploaded to the backend

This section contains instructions on how to create a neural network, upload it to the CPU backend, and execute.

> You can use this instruction for the development of application solutions.

To automatically upload a neural network to the backend and execute it:

1. In the directory of your project, create an application file with the .py extension where the function for creating and running a neural network is implemented.

2. In the application file, import the objects of the platform libraries required for the execution of the neural network.

   If necessary, define aliases by means of the `as` statement.

   An example of connecting library objects for executing a neural network with a population of BLIFAT neurons and a projection of delta synapses on a single-thread CPU backend:

   ```Python
   from knp.base_framework._knp_python_framework_base_framework import BackendLoader
   from knp.core._knp_python_framework_core import UID, BLIFATNeuronPopulation, DeltaSynapseProjection,
   SpikeMessage
   from knp.neuron_traits._knp_python_framework_neuron_traits import BLIFATNeuronParameters
   from knp.synapse_traits._knp_python_framework_synapse_traits import DeltaSynapseParameters, OutputType
   ```

3. Implement a neuron generator with the desired properties or use an existing generator from the framework library.

   An example of a BLIFAT neuron generator implementation:

   ```Python
   # A function is implemented that generates BLIFAT neurons.
   def neuron_generator(_): # type: ignore[no-untyped-def]
    return BLIFATNeuronParameters()
   ```

4. Implement a synapse generator with the desired properties or use an existing generator from the framework library.

   An example of a delta synapse generator implementation:

   ```Python
   # A function is implemented that generates delta synapses for the projection, which loops the
   # population output on itself.
   def synapse_generator(_): # type: ignore[no-untyped-def]
    return DeltaSynapseParameters(1.0, 6, OutputType.EXCITATORY), 0, 0


   # A function is implemented that generates delta synapses for the projection connected
   ```

```
# to an input channel.
def input_projection_gen(_): # type: ignore[no-untyped-def]
 return DeltaSynapseParameters(1.0, 1, OutputType.EXCITATORY), 0, 0
```

5. Create a function that creates the objects required for the neural network execution.

   An example of creating the main function:

```
Python

def main(): # type: ignore[no-untyped-def]
 # ...
```

6. In the created function, create a population object and pass the neuron generator to the constructor.

   An example of a population object creation:

```
Python

  # ...
  # Creates a population object with one BLIFATNeuron.
  population = BLIFATNeuronPopulation(neuron_generator, 1)
  # ...
```

7. In the created function, create a projection object that loops the output of the population to itself. Pass the associated population ID and the synapse generator to the constructor.

   An example of creating a projection that loops the output of a population to itself:

```
Python

  # ...
  # Creates a projection object with one delta synapse, which loops the output of the population
  # to itself.
  loop_projection = DeltaSynapseProjection(
  population.uid, population.uid, synapse_generator, 1
  )
  # ...
```

8. In the created function, create an input projection object. Pass the synapse generator and associated population ID to the constructor.

   An example of creating an input projection connected to the input channel and population:

```
Python

  # ...
  # Creates an input projection object with one delta synapse, which is assigned
  # null ID (UID (False)) The projection receives spikes from the input channel
  # and sends synaptic impact to the population object.
  input_projection = DeltaSynapseProjection(
  UID(False), population.uid, input_projection_gen, 1
  )

  # Stores the input projection UID in the input_uid variable.
  input_uid = input_projection.uid
  # ...
```

9. In the created function, load the backend instance and then load the created population and projections into it. Create a message endpoint.

   An example of loading the backend and creating a message endpoint:

```
Python

  # ...
  # Loads an instance of a single-threaded CPU backend from the local computer.
```

```
backend = BackendLoader().load(f'{pytestconfig.rootdir}/../lib/libknp-cpu-single-threaded-backend')

# Loads the created population into the backend instance.
backend.load_all_populations([population])
# Loads the created projections into the backend instance.
backend.load_all_projections([input_projection, loop_projection])

# Initialization of the backend.
backend._init()
# Creates the message endpoint object.
endpoint = backend.message_bus.create_endpoint()
# ...
```

10. In the created function, define the IDs of the input and output channels.

    An example of defining IDs of the input and output channels:

```
Python

# ...
# Creates arbitrary in_channel_uid and out_channel_uid identifiers for
# the input and output channels.
in_channel_uid = UID()
out_channel_uid = UID()
# ...
```

11. In the created function, subscribe the backend and the message endpoint to the messages from the input and output channels.

    An example of a backend and message endpoint subscription to `SpikeMessage`:

```
Python

# ...
# Subscribes the backend to the spikes from the input channel.
backend.subscribe(SpikeMessage, input_uid, [in_channel_uid])
# Subscribes the message endpoint to the spikes from the output channel.
endpoint.subscribe(SpikeMessage, out_channel_uid, [population.uid])
# ...
```

12. In the created function, execute the neural network. If necessary, create a list where to write the indices of the steps at which spikes are received by the output channel.

    An example of the executing 20 steps of a neural network:

```
Python

# ...
# Creates the results list that contains the indices of the spike steps.
results = []

for step in range(0, 20):
# Sends spikes at the following neural network execution steps: 0, 5, 10, and 15.
if step % 5 == 0:
# Displays the number of the step when the input channel sent the spike to the input projection.
print(f'STEP {step}')
message = SpikeMessage((in_channel_uid, step), [0])
endpoint.send_message(message)

backend._step()
messages_count = endpoint.receive_all_messages()
output = endpoint.unload_messages(SpikeMessage, out_channel_uid)
if len(output):
# Displays the number of the step when the spike arrived at the output channel.
print(f'STEP {step}')
```

Neural network execution steps

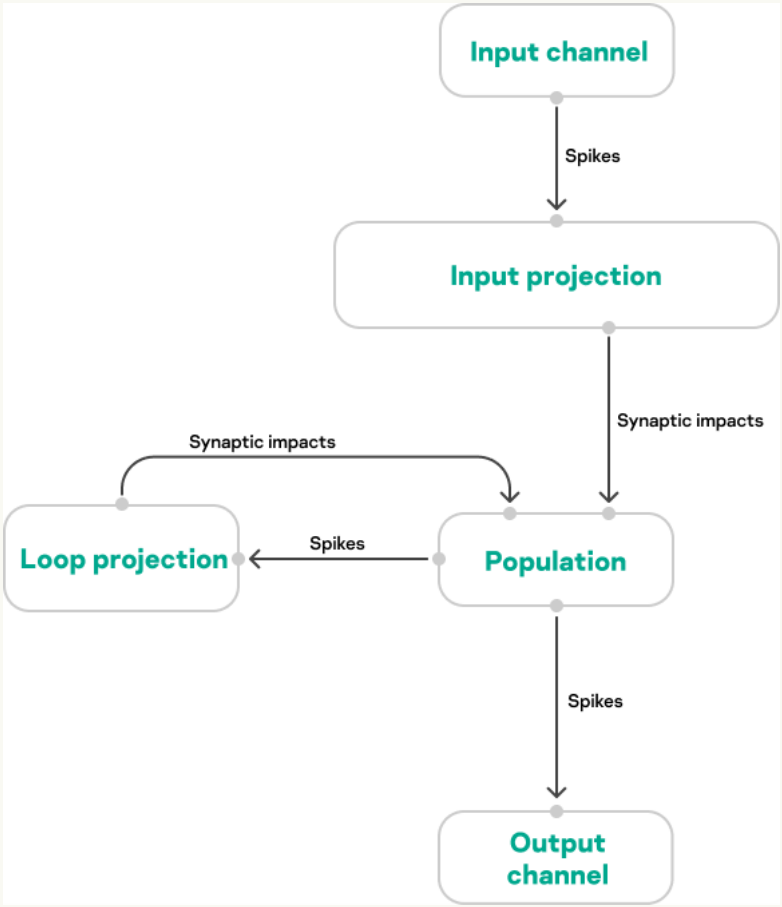
The neural network is executed in loops. The actions described in the loop are repeated every 5 steps of the network execution.

At the first step of the loop, the input channel sends the input projection spikes.

At the next step of the loop, the input projection computes synaptic impacts and sends them to the populations. After receiving a synaptic impact, the population sends spikes to the output channel and to the projection that loops the output of the population to itself.

After 6 steps of the loop are executed since the spikes sent by the input projection are received, the projection that loops the population output to itself calculates the synaptic impacts and sends the impact to the population.

The figure below shows a diagram of the neural network execution.



Neural network execution diagram

In this example, the neural network is executed in a loop with the following steps:

- *Step 0*. The input channel sends the input projection spikes. This step is repeated every 5 steps of the neural network execution loop.

- *Step 1*. The input projection computes synaptic impacts and sends them to the populations. After receiving a synaptic impact, the population sends spikes to the output channel and to the projection that loops the output of the population to itself. This step is repeated every 5 steps of the neural network execution.

- *Step 7*. The projection that loops the population output to itself computes the synaptic impacts after receiving the spikes sent 6 steps before and sends the impacts to the population. The population sends spikes to the output channel and to the projection that loops the output of the population to itself. This step is repeated every 5 steps of the neural network execution.

- *Step 13*. The projection that loops the population output to itself computes the synaptic impacts after receiving the spikes sent 6 steps before and sends the impacts to the population. The population sends spikes to the output channel and to the projection that loops the output of the population to itself. This step is repeated every 5 steps of the neural network execution.

- *Step 19*. The projection looping back the population inference calculates the synaptic influences after receiving the spikes sent in step 13, and transmits the population influences. The population sends spikes to the output channel and to the projection that loops the output of the population to itself.

13. Start the application.

A script for executing a neural network from ternary layers for image classification

You can use this script in the development of application solutions.

This section provides an example of creating, training and placing an image classification neural network, and its execution on the AltAI-1 neuromorphic processing unit. The neural network consists of `ANN2SNN` package ternary layers.

A script for executing a neural network from ternary layers for image classification consists of the following stages:

1. **Creating and training a neural network from ternary layers**

   Create a neural network using the ternary layers of the `ANN2SNN` package. Specify the `softmax` function as the activation function for the output layer. Prepare data for training the neural network. Convert the training data into binary form. Train the neural network and evaluate the training results.

2. **Neural network placement**

   Specify the AltAI-1 neuromorphic processing unit configuration settings in a YAML file. Place the neural network with the following command:

   ```
   placer -p <NN_file> -l <placer_log> -o <altai_config> -hw <hw_config>
   ```

   In this command:

   - `<NN_file>` is the path to the file with the trained neural network.
   - `<placer_log>` is the path to the file where the `placer` utility logs will be written.
   - `<altai_config>` is the path to the file in which the configuration will be saved for loading to the neuromorphic processor.
   - `<hw_config>` is the path to the file containing the neuromorphic processing unit configuration. In the absence of a `-hw` flag, the default neuromorphic processor configuration is used.

   Example:

   ```
   placer -p mnist.alt -l mnist_placer.log -o mnist.json
   ```

3. **Starting inference of the neural network**

Connect the backplane with the AltAI-1 neuromorphic processing unit to the computer via a `PCI-E` or `USB` interface.

Specify the configuration file and the component for the neural network execution. The neural network can be executed both on the AltAI-1 hardware processor and on its software emulator. Prepare the data for inference and convert it into binary form. Then start the inference.

Creating and training a neural network from ternary layers for image classification

This section contains instructions on how to create and train a neural network for image classification. The neural network is created using the ternary layers of the `ANN2SNN` package.

To create and train a neural network from ternary layers for image classification:

1. In the directory of your project, create an application file with the `.py` extension in which the neural network will be created and trained.

2. In the application file, import the modules of the libraries required for the creation and execution of the neural network.

   If necessary, define aliases by means of the `as` statement.

   An example of connecting objects for creating and executing a convolutional neural network:

   ```Python
   from knp_ann2snn.altainn import TernaryConv2D, TernaryDense
   from knp_ann2snn.altainn import heaviside
   from knp_ann2snn.altainn import Clip

   from tensorflow import keras
   from keras.models import Sequential, load_model
   from keras.layers import Flatten
   from keras.callbacks import ModelCheckpoint

   import tensorflow as tf
   ```

3. Create a neural network. Specify the `softmax` function as the activation function for the output layer.

   An example of creating a neural network with three convolutional layers:

   ```Python
   model = Sequential(
    [
    TernaryConv2D(1, 2, strides=2, activation=heaviside, input_shape=(28, 28, 1)),
    Flatten(),
    TernaryDense(10, activation="softmax")
    ]
   )
   model.compile(optimizer="adam", loss="categorical_crossentropy", metrics=["accuracy"])
   model.summary()
   ```

4. Prepare the training data.

   An example of preparing MNIST data for training:

   ```Python
   # Load MNIST data for training.
   (x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
   ```

```
# Encode target labels.
y_train = tf.keras.utils.to_categorical(y_train)
y_test = tf.keras.utils.to_categorical(y_test)

# Normalize input data.
x_train = x_train / 255
x_test = x_test / 255
# Convert input data to binary form.
x_train[x_train > 0.5] = 1
x_train[x_train <= 0.5] = 0

x_test[x_test > 0.5] = 1
x_test[x_test <= 0.5] = 0
# Replace data dimension (28, 28) with (28, 28, 1) for correct operation of convolutional layers.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.reshape(x_test.shape + (1,))
```

5. Create a callback function to save the neural network.

   An example of a callback function:

```
Python

# Save the trained neural network to mnist.h5 file.
model_chekpoint_callback = ModelCheckpoint(
 filepath="mnist.alt",
 monitor="accuracy",
 mode="max",
 save_best_only=True,
)
```

6. Train the neural network.

   An example of training a neural network:

```
Python

# Start training with the training data.
model.fit(x_train, y_train, epochs = 15, verbose = 1, callbacks=[model_chekpoint_callback])
# Load the trained model.
model = load_model(
 "mnist.alt",
 custom_objects = {
 "heaviside_mod": heaviside,
 "TernaryDense": TernaryDense,
 "TernaryConv2D": TernaryConv2D,
 "Clip": Clip
 }
)
# Evaluate the accuracy of the neural network.
print(f"Model accuracy: {model.evaluate(x_test, y_test, verbose = 1)[1]*100:.2f}%")
```

7. Start the application.

Running a neural network inference from ternary layers for image classification

This section contains instructions for starting the neural network inference for image classification on the AltAI-1 neuromorphic processor. First, create the neural network using `ANN2SNN` package ternary layers, train it and place to be executed on the neuromorphic processing unit.

To run the inference of a neural network from ternary layers for image classification:

1. Connect the backplane with the AltAI-1 neuromorphic processing unit to the computer via a `PCI-E` or `USB` interface.

2. In the directory of your project, create an application file with the `.py` extension in which the neural network inference will be implemented.

3. In the application file, import the modules of the libraries required for the inference of the neural network.

   If necessary, define aliases by means of the `as` statement.

   An example of connecting objects for inferencing a neural network:

```python
from knp_ann2snn import Altai
import tensorflow as tf
import numpy as np
from tqdm import tqdm
```

4. Create a function to calculate model accuracy.

   Example function for calculating model accuracy:

```python
def calculate_accuracy(array1,array2):

 correct_predictions = sum(a == b for a, b in zip(array1,array2))
 accuracy = correct_predictions / len(array1)

 return accuracy
```

5. Prepare the data for inference.

   An example of preparing MNIST data for inference:

```python
# Load the model.
altai = Altai()
# Specify the path to the configuration file for loading to the neuromorphic processor
# and the component for neural network execution:
# hw—for inference of the neural network on the AltAI-1 hardware processor
# gm—for inference of the neural network on a software emulator.
# If the activation function on the last layer is softmax, specify that.
altai.build("mnist.json", "gm", activation="softmax")


# Load MNIST data for inference.
(_, _), (x_test, y_test) = tf.keras.datasets.mnist.load_data()

# Normalize input data.
x_test = x_test / 255

# Convert input data to binary form.
x_test[x_test > 0.5] = 1
x_test[x_test <= 0.5] = 0

x_test = x_test.reshape(x_test.shape + (1,))
```

6. Start the inference.

   An example of inference:

```python
pred = []

for i in tqdm(range(x_test.shape[0])):
 # Convert input vector to spikes.
 altai.prepare_spikes(x_test[i])
```

```
    # Start tick signal generation.
    altai.start_ticks(2)
    # Receive spikes from the neural network.
    spikes_idx = altai.get_spikes()
    # We use index -1 because we are looking for the result of the final tick.
    predicted_class = spikes_idx[-1]
    pred.append(predicted_class)

# Evaluate the accuracy of the neural network.
print("Accuracy: ", calculate_accuracy(y_test, pred)* 100,"%")
```

7. Start the application.

A script for executing a regression model with training blocks that encode and decode data

You can use this script in the development of application solutions.

This section provides an example of creating, training and placing a regression model, and its execution on the AltAI-1 neuromorphic processing unit.

A script for executing a regression model with training blocks that encode and decode data consists of the following stages:

1. **Creating and training a regression model**

   Create a regression model consisting of an encoding block, a neural network, and a decoding block. Prepare the data for training the model. Train the model and evaluate the training results.

2. **Neural network placement**

   Specify the AltAI-1 neuromorphic processing unit configuration settings in a YAML file. Place the neural network with the following command:

   `placer -p <NN_file> -l <placer_log> -o <altai_config> -hw <hw_config>`

   In this command:

   - `<NN_file>` is the path to the file with the trained neural network.
   - `<placer_log>` is the path to the file where the `placer` utility logs will be written.
   - `<altai_config>` is the path to the file in which the configuration will be saved for loading to the neuromorphic processor.
   - `<hw_config>` is the path to the file containing the neuromorphic processing unit configuration. In the absence of a `-hw` flag, the default neuromorphic processor configuration is used.

   Example:

   `placer -p snn_sin.keras -l sin_placer.log -o sin.json`

3. **Running the regression model inference**

   Connect the backplane with the AltAI-1 neuromorphic processing unit to the computer via a `PCI-E` or `USB` interface.

Specify the configuration file and the component for the neural network execution. The neural network can be executed both on the AltAI-1 hardware processor and on its software emulator. Run the inference of the regression model.

Creating and training a regression model

This section provides instructions on how to create and train a regression model with trainable blocks that encode and decode data. The model is created using the ternary layers of the `ANN2SNN` package.

You can use this instruction in the development of application solutions.

To create and train a regression model:

1. In the directory of your project, create an application file with the `.py` extension in which the model will be created and trained.

2. In the application file, import the modules of the libraries required for the creation and training of the model.

   If necessary, define aliases by means of the `as` statement.

   An example of connecting objects to create and train a regression model:

   ```Python
   from knp_ann2snn.altainn import TernaryDense
   from knp_ann2snn.altainn import heaviside
   from knp_ann2snn.altainn import Clip

   from tensorflow import keras
   from keras.models import Sequential, load_model
   from keras.callbacks import ModelCheckpoint
   from keras.layers import Dense


   import matplotlib.pyplot as plt
   import numpy as np
   ```

3. Create a sine wave and prepare the data for training.

   An example of data preparation:

   ```Python
   # Define the number of elements in the data set.
   N = 4000
   # Define the index up to which the elements of the data set will be used for training.
   train_idx = 3000
   # Define the index up to which elements, starting from the train_idx index, form test data.
   test_idx = 4000
   # Define the number of elements in a batch.
   elements_in_batch = 10
   # Define the dimension of the output of the encoding block.
   encoder_output_shape = 25
   # Define the dimension of the output of the neural network.
   snn_output_shape = 35
   # Define the dimension of the output of the decoding block.
   decoder_output_shape = 1


   # Create a sine wave.
   x = np.arange(N)
   arr = np.sin(x)
   ```

```
# Generate training data based on elements_in_batch values.
train_range = range(train_idx - elements_in_batch - 1)
x_train = np.array([arr[i : i + elements_in_batch] for i in train_range])
y_train = np.array([[arr[i + elements_in_batch + 1]] for i in train_range])


# Generate test data based on elements_in_batch values.
test_range = range(train_idx, test_idx - elements_in_batch - 1)
x_test = np.array([arr[i : i + elements_in_batch] for i in test_range])
y_test = np.array([[arr[i + elements_in_batch + 1]] for i in test_range])
```

4. Create a regression model. To do this, create an encoding block, a spiking neural network, and a decoding block.

   An example of creating a regression model:

```
Python

# Create a single-layer coding block that converts data to spikes.
encoder = Sequential(
 [
 Dense(
 encoder_output_shape,
 activation=heaviside,
 input_shape=(elements_in_batch,)
 )
 ]
)


# Create a spiking neural network.
snn = Sequential(
 [
 TernaryDense(
 snn_output_shape,
 activation=heaviside,
 input_shape=(encoder_output_shape,),
 use_bias=False
 )
 ]
)


# Create a single-layer decoding block that converts spikes to a target value.
decoder = Sequential(
 [
 Dense(
 decoder_output_shape,
 activation="linear",
 input_shape=(snn_output_shape,)
 )
 ]
)

# Create a model with a coding block, neural network, and decoding block.
model = Sequential([encoder, snn, decoder])
model.compile(optimizer="adam", loss="mse")
```

5. Train the model.

   An example of model training:

```
Python

# Train the model on the training data.
checkpoint = ModelCheckpoint(
 "sin_model.keras",
 monitor="val_loss",
 verbose=1,
 save_best_only=True,
 mode="min"
)
model.fit(x_train, y_train, epochs=50, validation_data=(x_test, y_test), callbacks=[])

model.save("sin_model.keras")

# Load the model.
```

```
model = load_model(
 "sin_model.keras",
 custom_objects={
 "heaviside_mod": heaviside,
 "Clip": Clip,
 "TernaryDense": TernaryDense,
 },
)

# Obtain predictions.
predict_test = model.predict(x_test)
predict_test = predict_test.flatten()

# Plot graphs based on predicted and target values.
plt.plot(predict_test, "b")
plt.plot(y_test, "r", alpha=0.5)
plt.savefig("sin_train.png")
```

6. Save the coding block, neural network, and decoding block.

   An example of saving:

```
Python

# Save the encoding block to the encoder_sin.keras file.
model.layers[0].save("encoder_sin.keras")
# Save the neural network to the snn_sin.keras file.
model.layers[1].save("snn_sin.keras")
# Save the decoding block to the decoder_sin.keras file.
model.layers[2].save("decoder_sin.keras")
```

7. Start the application.

Running the regression model inference

This section provides instructions for running inference of a regression model with an encoding and decoding block. First, create, train, and place the model for execution on a neuromorphic processing unit.

To run the inference of the regression model:

1. Connect the backplane with the AltAI-1 neuromorphic processing unit to the computer via a `PCI-E` or `USB` interface.

2. In the directory of your project, create an application file with the `.py` extension, in which the model inference will be implemented.

3. In the application file, import the modules of the libraries required for the inference of the model.

   If necessary, define aliases by means of the `as` statement.

   An example of connecting objects for inferencing a model:

```
Python

from knp_ann2snn.altainn import TernaryDense
from knp_ann2snn.altainn import heaviside
from knp_ann2snn.altainn import Clip
from knp_ann2snn.python_altai import Altai

from keras.models import load_model
import matplotlib.pyplot as plt

from tqdm import tqdm
import numpy as np
```

4. Create a sine wave and prepare the data for inference.

   An example of data preparation:

```Python
# Define the number of elements in the data set.
N = 4000
# Define the index up to which the elements of the data set will be used for training.
train_idx = 3000
# Define the index up to which elements, starting from the train_idx index, form test data.
test_idx = 4000
# Define the number of elements in a batch.
elements_in_batch = 10
# Define the dimension of the output of the encoding block.
encoder_output_shape = 25
# Define the dimension of the output of the neural network.
snn_output_shape = 35
# Define the dimension of the output of the decoding block.
decoder_output_shape = 1


# Create a sine wave.
x = np.arange(N)
arr = np.sin(x)


# Generate test data based on elements_in_batch values.
test_range = range(train_idx, test_idx - elements_in_batch - 1)
x_test = np.array([arr[i : i + elements_in_batch] for i in test_range])
y_test = np.array([[arr[i + elements_in_batch + 1]] for i in test_range])
```

5. Load the model, encoding and decoding block.

   An example of loading a model, encoding and decoding block:

```Python
altai = Altai()
# Specify the path to the configuration file for loading to the neuromorphic processor
# and the component for neural network execution:
# hw—for inference of the neural network on the AltAI-1 hardware processor
# gm—for inference of the neural network on a software emulator.
altai.build("sin.json","gm")


encoder = load_model(
 "encoder_sin.keras",
 custom_objects={
 "heaviside_mod": heaviside
 }
 )

snn_model = load_model(
 "snn_sin.keras",
 custom_objects={
 "TernaryDense": TernaryDense,
 "heaviside_mod": heaviside,
 "Clip": Clip
 }
 )

decoder = load_model("decoder_sin.keras")
```

6. Start the inference.

   An example of running inference:

```Python
predict = np.zeros(len(x_test))

# Start a prediction cycle.

for i in tqdm(range(len(x_test))):
```

```
# Predict the coding block.
encoded = encoder.predict_on_batch(x_test[i:i+1])

# Predict the neural network.
altai.prepare_spikes(encoded)
altai.start_ticks(1)

spikes_idx = altai.get_spikes()
spikes_idx = spikes_idx[spikes_idx!=-2147483647]
spikes = np.zeros(snn_output_shape)
spikes[spikes_idx] = 1

# Predict the decoding block.
decoded = decoder.predict_on_batch(spikes.reshape(1,-1))

# Update the array of predictions.
predict[i] = decoded
predict = predict.reshape(len(predict))
y_test = y_test.reshape(len(y_test))
```

7. Evaluate the inference results. To do this, you can compare the sine wave values with the model predictions on the graph.

   An example of constructing a graph to evaluate inference results:

   Python

   ```
   # Plot the graphs.
   plt.plot(predict, "b")
   plt.plot(y_test, "r", alpha=0.5)
   plt.savefig("sin_inference.png")
   plt.show()
   ```

8. Start the application.

Using the Kaspersky Neuromorphic Platform API

Kaspersky Neuromorphic Platform has an Application Programming Interface (API) that provides access to platform functions.

Using the Kaspersky Neuromorphic Platform API, you can do the following:

- Get data about used devices and backends
- Get data about the `core` library objects
- Get data about the platform frontend objects
- Manage `core` library objects
- Create and train neural networks
- Manage neural network hyperparameters
- Created new neuron models
- Create new topologies of neural networks
- Implement application solutions



Open C++ API reference guide (English)

Participation in the development of the platform

You can participate in the development of the Kaspersky Neuromorphic Platform. To do this, you need to get the source code from the platform repository.

To participate in the development of the platform:

1. Open the terminal and use the `cd` command to go to the working directory.

   You can use any other tool for working with Git. For example, in Windows, you can use Visual Studio.

2. To obtain the source code for the platform, you must perform the following actions:

   a. At a command line, run the following command to download the platform repository to the device:

   `git clone <platform_repository>`

   `<platform_repository>` is the platform repository web address received from Kaspersky experts.

   b. Install Python packages by running the following command:

   ```
   pip3 -r requirements.txt
   ```

   c. If necessary, download the remote platform repository to your local repository. You can use the following command for this:

   ```
   git pull
   ```

3. In the command line, run the following command to create the branch where you will be making changes to the platform code:

   `git checkout -b <branch_name>`

4. Make the changes to the platform code.

5. Commit the changes.

   For example, you can use the following commands to publish changes:

   ```
   git add <publishing_file_name>
   git commit -m "<publish_message>"
   git push origin <branch_name>
   ```

6. Publish the changes to the platform repository.

Platform licensing

The Kaspersky Neuromorphic Platform Terms of Use is a binding agreement between you and Kaspersky that stipulates the terms and conditions on which you may use the platform. Kaspersky Neuromorphic Platform is available under the Apache 2.0 license.

The Kaspersky Neuromorphic Platform Terms of Use apply to the platform source code and the following packages:

- `knp-cpu-single-threaded-backend_<knp_version>_amd64.deb`: a deb package with CPU single-threaded backend binaries
- `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb` a deb package with CPU multi-threaded backend binaries
- `knp-cpp-framework_<knp_version>_amd64.deb`: a deb package with C++ framework binaries
- `knp-cpu-multi-threaded-backend-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`
- `knp-cpu-single-threaded-backend-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`
- `knp-cpp-framework-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-cpp-framework_<knp_version>_amd64.deb`
- `knp-python3-framework_<knp_version>_amd64.deb`: a deb package containing a Python framework for an `AltAI SNN` backend.
- `knp-examples_<knp_version>_amd64.deb`: a deb package with Kaspersky Neuromorphic Platform usage examples
- `knp-documentation_<knp_version>_all.deb`: a deb package with the API reference guide
- `knp-<knp_version>-py3-none-any.whl`: a whl package containing a Python framework for an `AltAI SNN` backend.

  where `<knp_version>` is the Kaspersky Neuromorphic Platform version.

You can review the Kaspersky Neuromorphic Platform Terms of Use by reading the `LICENSE.txt` located in the following directories:

- The root of the platform repository
- `/usr/share/doc/knp`, after installing `knp-cpp-framework_<knp_version>_amd64.deb`
- `/usr/lib/python<Python version>/knp`, after installing `knp-<knp_version>-py3-none-any.whl`
- The root of the Docker image `knp-build-image` for building the platform or an application.

  where:

  - `<Python version>` is the installed Python version.
  - `<knp_version>` is the Kaspersky Neuromorphic Platform version.

The AltAI-1 backends and `ANN2SNN` package Terms of Use constitute a binding agreement between you and Motiv NT, which sets out the terms of using the following packages:

- `knp-altai-backend_<motiv_version>_amd64.deb`: a deb package with `AltAI SNN` backend binary for the AltAI-1 neuromorphic processor
- `knp-altai-backend-dev_<motiv_version>_amd64.deb`: a dev-deb package for `knp-altai-backend_<motiv_version>_amd64.deb`
- `knp_ann2snn.deb`: a deb package containing an `ANN2SNN` package and `AltAI ANN2SNN` backend binaries
- `knp_ann2snn-<motiv_version>-py3-none-any.whl`: a whl package containing an `ANN2SNN` package and `AltAI ANN2SNN` backend binaries to install on an x86-based computer
- `knp_ann2snn_aarch64-<motiv_version>-py3-none-any.whl`: a whl package with an `ANN2SNN` package and `AltAI ANN2SNN` backend binaries to install on an ARM-based computer

  where `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package.

You can review the Terms of Use as follows:

- While installing the deb packages
- In the `LICENSE.txt` located in `/usr/share/doc/altai-backend/`, after installing `knp-altai-backend_<motiv_version>_amd64.deb` or `knp-altai-backend_<motiv_version>_amd64.deb`
- In the `LICENSE.txt` located in `/usr/share/doc/knp_ann2snn/`, after installing `knp_ann2snn.deb`
- In the `LICENSE.txt` located in `/usr/lib/python<Python version>/site-packages/knp_ann2snn/`, after installing any of the whl packages

  where:

  - `<Python version>` is the installed Python version.
  - `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package.

Carefully read the Terms of Use before you start working with Kaspersky Neuromorphic Platform.

If you do not agree with the Terms of Use, uninstall and do not use Kaspersky Neuromorphic Platform. The package installation process will be interrupted unless you accept the AltAI-1 backends and `ANN2SNN` package Terms of Use when installing the deb packages.

About data provision

Kaspersky Neuromorphic Platform does not request, store, or process information related to personal data.

Appendix. Settings and an example YAML file containing the configuration of the AltAI-1 neuromorphic processing unit

Configuration file settings must be defined to run the neural network on an `AltAI ANN2SNN` backend.

The configuration file specifies the number of VLSIs, cores, neurons, and synapses in the AltAI-1 neuromorphic processing unit. The configuration file contains the following settings:

- `chip` includes a `cores` setting. This `cores` value sets the number of cores in one VLSI.
- `core` is a group of the following settings that describe the core:
  - `neurons` is the number of neurons in the core.
  - `axons` is the number of input lines.
  - `synapses` is the number of synapses in the core.
- `row` includes a `chips` setting. This `chips` value sets the number of VLSIs in one module of the AltAI-1 neuromorphic processing unit backplane.
- `col` includes a `chips` setting. This `chips` value sets the number of modules in the AltAI-1 neuromorphic processing unit backplane.
- `not_working_chips` is an array with the numbers of VLSIs that will not participate in the execution of the neural network on the neuromorphic processing unit—for example, if these VLSIs are missing on the backplane modules.

  This setting must be configured by a qualified Customer engineer, a Kaspersky employee, or a certified integration partner.

Below is an example YAML file containing the default configuration of the AltAI-1 neuromorphic processing unit. In the example below, the neuromorphic processing unit backplane consists of four modules with eight VLSIs in each. Each VLSI contains 16 cores, each of these with 512 neurons, synapses, and input lines.

hw_config.yaml

```yaml
chip:
  cores: 16
core:
  neurons: 512
  axons: 512
  synapses: 512
row:
  chips: 8
col:
  chips: 4
not_working_chips: []
```

Glossary

AltAI ANN2SNN backend

A backend for training and execution of artificial neural networks (ANN) converted to spiking neural networks running on an AltAI-1 neuromorphic processor.

AltAI SNN backend

A backend for execution of spiking neural networks running on an AltAI-1 neuromorphic processor.

Artificial Neural Network (ANN)

A neural network based on artificial neurons that perform an instantaneous summation of input signals represented as numbers and weighted according to the weights of input connections, and generate an output numerical signal according to a given activation function.

Backend

The software part of the platform that provides a universal interface to the computer and implements the neuron and synapse functions, as well as the functions of their modifiers in terms of the computer.

Computer

A device or code that directly executes a network.

Connection

Connection of a synapse with a presynaptic or postsynaptic neuron.

Cycle

The time slice of the clock signal during which the kernel computer performs one operation.

Dense sequence of spikes (DSS)

A sequence of unforced spikes generated by a neuron, where all adjacent spikes are separated by the time intervals by no more than ISImax, where ISImax is the maximum allowed distance between spikes.

DSS period

The time interval between the moment when the Hebbian plasticity starts before the first spike in the DSS and one of the following moments: when the maximum allowable distance (ISImax) is reached after the last spike in the DSS, when a forced activation occurs, or when the spike arrives at the dopamine synapse.

External environment

A set of elements external to the Kaspersky Neuromorphic Platform with which the platform interacts.

Framework

The software part of the platform, which determines its structure and enables backend management. The Kaspersky Neuromorphic Platform implements a C++ framework.

Modifier

Code that modifies the behavior of a neuron or synapse, for example, adds synaptic plasticity.

Neuron

A neural network node with a set of attributes. Based on the neuron attributes and the sequence of input signals, a certain mathematical function is calculated. The result of the function is a synaptic impact or its absence.

Neuron function

A system that describes and modifies the neuron behavior. The function results in a spike or its absence.

Population

A container containing a set of neurons and their function based on the neuron type.

Projection

A container containing a set of synapses of the same type connecting neurons of the presynaptic and postsynaptic populations.

Spike

Short-term action potential generated by a neuron in a population as a result of the neuron attribute changes.

Spiking Neural Network (SNN)

A neural network based on artificial neurons that perform temporal summation of input signals represented by spikes and weighted according to the weights of input connections, and generate an output pulse according to a certain threshold-based activation function.

Synapse

Neuron junction with the main function to transmit a signal from one neuron to another. Each neuron can be connected to multiple synapses.

Synapse function

A system that simulates the synapse behavior. The function results in a synaptic impact or its absence.

Synapse weight

The value of the signal that the synapse sends to the neurons of the associated population.

Synaptic impact

A value generated by a synapse for changing the values of the neuron attributes.

Tick

The time slice during which the function of all neurons in the core is calculated. During a tick, all spikes generated by the cores or received on the input lines of neurons are delivered to the target neurons.

Information about third-party code

Information about third-party code in Kaspersky Neuromorphic Platform can be found in the `NOTICE.txt` files located in the following directories:

- The root of the platform repository, if working with the platform source code
- The root of the Docker image, if working with the Docker image `knp-build-image`.
- `/usr/share/doc/knp`, after installing `knp-cpp-framework_<knp_version>_amd64.deb`
- `/usr/lib/python<Python version>/knp`, after installing `knp-<knp_version>-py3-none-any.whl`

  where:

  - `<Python version>` is the installed Python version.
  - `<knp_version>` is the Kaspersky Neuromorphic Platform version.

Information about third-party code in packages provided by Motiv NT can be found in the `NOTICE.txt` located in the following directories:

- `/usr/share/doc/altai-backend/` after installing the `knp-altai-backend_<motiv_version>_amd64.deb` or `knp-altai-backend-dev_<motiv_version>_amd64.deb` packages.
- `/usr/share/doc/knp_ann2snn/`, after installing `knp_ann2snn.deb`
- `/lib/python<Python version>/site-packages/knp_ann2snn/` after installing the `knp_ann2snn-<motiv_version>-py3-none-any.whl` or `knp_ann2snn_aarch64-<motiv_version>-py3-none-any.whl` packages.

  where:

  - `<Python version>` is the installed Python version.
  - `<motiv_version>` is the version of the backends for the AltAI-1 neuromorphic processor and the `ANN2SNN` package.

Trademark notices

Registered trademarks and service marks are the property of their respective owners.

Apache is either a registered trademark or a trademark of the Apache Software Foundation.

Ubuntu and LTS are registered trademarks of Canonical Ltd.

Docker and the Docker logo are trademarks or registered trademarks of Docker, Inc. in the United States and/or other countries. Docker, Inc. and other parties may also have trademark rights in other terms used herein.

TensorFlow and any associated designations are trademarks of Google LLC.

Intel and Core are trademarks of Intel Corporation in the U.S. and/or other countries.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft, Visual Studio and Windows are trademarks of the Microsoft group of companies.

JavaScript is a registered trademark of Oracle and/or its affiliates.

Python is a trademark or registered trademark of the Python Software Foundation.

Debian is a registered trademark of Software in the Public Interest, Inc.