

The Kaspersky logo is displayed in a bold, lowercase, sans-serif font. It is positioned within a white, rounded rectangular area that is centered on a background with a green-to-teal gradient and abstract, flowing shapes.

kaspersky

Kaspersky Neuromorphic Platform

© 2025 AO Kaspersky Lab

Contents

[About Kaspersky Neuromorphic Platform](#)

[Distribution kit](#)

[Distribution kit for the development of application solutions in Linux](#)

[Distribution kit for the development of application solutions in Python for Linux](#)

[Distribution kit for working with the platform source code](#)

[Hardware and software requirements](#)

[What's New](#)

[Key concepts of the Kaspersky Neuromorphic Platform](#)

[Neurons of a spiking neural network](#)

[Neuron based on the AltAILIF model](#)

[Neuron based on the BLIFATNeuron model](#)

[Neuron based on the SynapticResourceSTDPBLIFATNeuron model](#)

[Synapses of a spiking neural network](#)

[Synapse based on the AdditiveSTDPDeltaSynapse model](#)

[Synapse based on the DeltaSynapse model](#)

[Synapse based on the SynapticResourceSTDPDeltaSynapse model](#)

[Altai-2 neuromorphic processor](#)

[Kaspersky Neuromorphic Platform backends](#)

[Backends for execution of spiking neural networks](#)

[Backends for training and execution of artificial neural networks](#)

[Neural network description format](#)

[Data formats processed by the platform](#)

[Kaspersky Neuromorphic Platform architecture](#)

[Changing the configuration settings of the software emulator](#)

[General operating principles of Kaspersky Neuromorphic Platform](#)

[Data exchange in Kaspersky Neuromorphic Platform](#)

[CPU backend operation](#)

[Processing messages by subscription](#)

[Installing and uninstalling the platform](#)

[Installing deb packages](#)

[Installing Python development packages](#)

[Uninstalling the platform](#)

[Scenario for building a platform project or an application solution in C++](#)

[Downloading and unpacking the platform source code archive](#)

[Defining the Boost_ROOT setting in Windows](#)

[Specifics of building the platform with a Python framework](#)

[Configuring a build of a platform or an application solution in Windows](#)

[Configuring the project build settings for the application solution in C++](#)

[C++ platform components](#)

[neuron_traits library](#)

[synapse_traits library](#)

[core library](#)

[messaging namespace](#)

[tags namespace](#)

[Backend class](#)

[BaseData structure](#)

[continuously_uid_generator class](#)

[Device class](#)

[MessageBus class](#)

[MessageEndpoint class](#)

[Population class](#)

[Projection class](#)

[Subscription class](#)

[TagMap class](#)

[UID class](#)

[uid_hash\(\).function](#)

[meta library](#)

[devices namespace](#)

[C++ framework](#)

[coordinates namespace](#)

[data_processing namespace](#)

[inference_evaluation namespace](#)

[input namespace](#)

[modifier namespace](#)

[monitoring namespace](#)

[normalization namespace](#)

[output namespace](#)

[population namespace](#)

[projection namespace](#)

[sonata namespace](#)

[storage namespace](#)

[synchronization namespace](#)

[BackendLoader class](#)

[Model class](#)

[ModelExecutor class](#)

[ModelLoader class](#)

[Network class](#)

[Solving common tasks in C++](#)

[Adding a new neuron type](#)

[Adding a new synapse type](#)

[Preparing the model and executing it on the backend](#)

[Execution of a neural network created from projections and populations](#)

[Python platform components](#)

[neuron_traits module](#)

[synapse_traits module](#)

[core module](#)

[messaging module](#)

[Backend class](#)

[BaseData structure](#)

[continuously_uid_generator class](#)

[Device class](#)

[MessageBus class](#)

[MessageEndpoint class](#)

[Set of Population classes](#)

[Set of Projection classes](#)

[Set of Subscription classes](#)

[TagMap class](#)

[UID class](#)

[uid_hash class](#)

[UUID class](#)

[Python framework](#)

[ANN2SNN package](#)

[Ternary layers of the ANN2SNN package](#)

[input module](#)

[output module](#)

[sonata module](#)

[BackendLoader class](#)

[ImpactMessageObserver class](#)

[Model class](#)

[ModelExecutor class](#)

[ModelLoader class](#)

[Network class](#)

[SpikeMessageObserver 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](#)

[Troubleshooting](#)

[The platform is not being built with the Python framework](#)

[The compiler or build system cannot find the Boost library in Windows](#)

[Applications](#)

[The structure of the file describing neuron groups](#)

[The structure of the file describing synapse groups](#)

[Structure and example of a basic format file](#)

[Model quality evaluation results transmitted in CSV format](#)

[Formulas for calculating model quality evaluation metrics](#)

[Settings and an example YAML file containing the configuration of the AltAI-2 neuromorphic processing unit](#)

[Build configuration parameters specific to the Kaspersky Neuromorphic Platform](#)

[Presets for building the platform](#)

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](#)

[Winner Takes ALL \(WTA\) mechanism](#)

[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.
- 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 that use neuromorphic processors.

To solve the above tasks, you can use the C++ and Python® languages. The platform supports CPUs, GPUs with CUDA support, as well as the AltAI-2 neuromorphic processing unit designed for power-efficient execution of neural networks on various classes of smart devices.

Distribution kit

This section provides information about the distribution kits of Kaspersky Neuromorphic Platform. This platform has multiple options for distribution kits. 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](#).

Distribution kit for the development of application solutions in Linux

The distribution kit can be used for developing application solutions in a Linux operating system.

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 dynamic library.
 - `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`: a deb package with a CPU multi-threaded backend dynamic library.
 - `knp-gpu-cuda-backend_<knp_version>_amd64.deb`: a deb-package with a backend library for GPUs with support for CUDA technology.
 - `knp-cpp-framework_<knp_version>_amd64.deb`: a deb package with a C++ framework dynamic library, 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:
 - `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-gpu-cuda-backend-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-gpu-cuda-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 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` is the deb package containing the API reference.
- `knp-unspecified_<knp_version>_amd64.deb`: a deb package that includes CMake service.
- Packages provided by Motiv NT.

- `knp-altai-backend_<motiv_version>_amd64.deb`: a deb package that contains an [AltAI SNN backend](#) dynamic library.
- `knp-altai-backend-dev_<motiv_version>_amd64.deb`: a dev-deb package for `knp-altai-backend_<motiv_version>_amd64.deb`.
- `Knp_ann2snn.deb` is a deb package with the ANN2SNN package, [AltAI ANN2SNN backend](#) dynamic library for the [AltAI-2 neuromorphic processor](#), and a component for placing the neural network on AltAI-2.

where:

- `<knp_version>` is the Kaspersky Neuromorphic Platform version.
- `<motiv_version>` is the version of the backends for the AltAI-2 neuromorphic processor and the ANN2SNN package.

Distribution kit for the development of application solutions in Python for Linux

The distribution kit can be used for developing application solutions in Python for Linux operating system.

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.
- Packages provided by Motiv NT:
 - `Knp_ann2snn.deb` is a deb package with the ANN2SNN package, `AltAI ANN2SNN backend` dynamic library for the [AltAI-2 neuromorphic processor](#), and a component for placing the neural network on AltAI-2.
 - `knp_ann2snn_x86_64-<motiv_version>-py3-none-any.whl`: a whl package with an ANN2SNN package, `AltAI ANN2SNN backend` dynamic library for the [AltAI-2 neuromorphic processor](#), and a component for placing the neural network on AltAI-2.
This package can be installed on an x86-based computer.
 - `Knp_ann2snn_aarch64-<motiv_version>-py3-none-any.whl`: a whl package with the ANN2SNN package and the `AltAI ANN2SNN backend` dynamic library for the [AltAI-2 neuromorphic processor](#).
This package can be installed on an ARM-based computer.

where:

- `<knp_version>` is the Kaspersky Neuromorphic Platform version.
- `<motiv_version>` is the version of the backends for the AltAI-2 neuromorphic processor and the ANN2SNN package.

Distribution kit for working with the 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
- `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
- GPU with CUDA support: NVIDIA GPU using Pascal or higher
- Neuromorphic processor (when using the `AltAI_SNN` or `AltAI_ANN2SNN` backends): AltAI-2
- 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 or later

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. For example, for Visual Studio 2022, the compiler version is 14.3.

- CMake 3.25 or later

- CUDA Toolkit 12.8

This software must be installed to run on a computer with an NVIDIA GPU.

- ZeroMQ 4.3.5

This software must be installed to run on a computer with a CPU or NVIDIA GPU.

- Before installing platform component deb packages for an [AltAI ANN2SNN backend](#), make sure that the device has the following software:

- TensorFlow 2.16.2 or later
- Keras 3.12 or later
- NumPy 2.16.2 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 3.12 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 2.16.2
 - 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 2.16.2

Kaspersky Neuromorphic Platform uses the following third-party libraries:

Boost [↗](#)

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

What's New

Kaspersky Neuromorphic Platform 2.0 adds the following features and improvements:

- [Distribution kit for developing Linux applications](#): added packages containing a backend library for GPUs with CUDA support technology.
- Backends:
 - [CUDABackend](#): added an interface for the GPU backend.
 - [SingleThreadedCPUBackend](#) backend: added neuron support based on the `AltAILIF` model.
 - [MultiThreadedCPUBackend](#) backend: added neuron support based on the `SynapticResourceSTDPBLIFATNeuron` model and synapse based on the `SynapticResourceSTDPDeltaSynapse` model.
 - Optimized CPU backend operation.
- [C++ components](#):
 - [Core](#) library: extended the functionality for the [Backend](#), [MessageBus](#), [MessageEndpoint](#), [Population](#), [Projection](#), [Subscription](#), and [TagMap](#) classes.
 - [Framework for C++](#): extended the functionality of the C++ framework:
 - [data_processing](#) namespace: added a namespace that provides functionality for processing classification model training and testing data.
 - [inference_evaluation](#) namespace: Added a namespace that provides functionality for evaluating model quality.
 - [modifier](#) namespace: added a namespace that provides message handlers.
 - [normalization](#) namespace: added a namespace that provides functions and functors for normalizing the parameters of neurons and synapses in neural networks.
 - [monitoring](#) namespace: added the `model` namespace with model monitoring functions.
 - [projection](#) namespace: added functions to generate synapses and synapse projections.
 - [ModelExecutor](#) class: added methods.
 - [neuron_traits](#) library: extended the neuron interfaces for the [BLIFATNeuron](#) and [SynapticResourceSTDPBLIFATNeuron](#) models.
- [Components for use in Python](#): added methods for the [ModelExecutor](#) and [UID](#) classes.
- [synapse_traits](#) library (for use in C++ and Python): extended the interface of the synapse based on the `SynapticResourceSTDPDeltaSynapse` model.
- Example [mnist-learn](#) is an added example of training a neural network on images of handwritten numbers and their labels from the MNIST database and its execution.

Key concepts of the Kaspersky Neuromorphic Platform

This section contains detailed definitions of the key concepts used in the Kaspersky Neuromorphic Platform.

Neurons of a spiking neural network

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. When modeling some neurons, the leak mechanism intended for the relaxation of the neural potential can be used.

Kaspersky Neuromorphic Platform supports sets of properties for the following neurons:

- Neuron based on the [AltAILIF](#) model
- Neuron based on the [BLIFATNeuron](#) model ("BLIFAT neuron")
- Neuron based on the [SynapticResourceSTDPBLIFATNeuron](#) model

Neuron based on the AltAILIF model

The neuron based on the [AltAILIF](#) model is a leaky integrator neuron (LIF 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 a leakage value, 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

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 addition to the synaptic impact, the value of the membrane potential can change due to stochastic stimulation. When stochastic stimulation is applied, a random number is added to the neuron potential each cycle.

The threshold value of the membrane potential of a neuron depends on the following parameters:

- Baseline threshold value. After the spike is generated, the membrane potential tends toward the baseline value.
- Dynamic component of the threshold value. After the spike is generated, the dynamic component of the threshold value changes according to 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, the interval for spike generation, and the stochastic stimulation parameters 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.

In addition to the baseline threshold value and the dynamic component, calculation of the threshold value of the neuron membrane potential also takes into account an additional threshold value. The additional threshold value, in turn, depends on the weights and activity of the synapses.

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 a connected synapse 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](#).

Synapses of a spiking neural network

The junction of [neurons](#) of the spiking neural network where the signal is transmitted is called a *synapse*. 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.

Kaspersky Neuromorphic Platform supports sets of properties for the following synapses:

- Synapse based on the [AdditiveSTDPDeltaSynapse](#) model
- Synapse based on the [DeltaSynapse](#) model (hereinafter also referred to as delta synapse)
- Synapse based on the [SynapticResourceSTDPDeltaSynapse](#) model

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 DeltaSynapse model

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 *ISI_{max}*, where *ISI_{max}* 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 *ISI_{max}* 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.
- $ISI_{\max}$ 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.
- $w_{\min}$ corresponds to a random minimum weight.
- $w_{\max}$ corresponds to a random maximum weight.

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

Formula for calculating the weight change depending on the synaptic resource

Altai-2 neuromorphic processor

The *AltAI-2 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-2 neuromorphic processor is intended for the execution of [spiking neural networks](#) created using [platform components](#) or using ternary layers of the [ANN2SNN package](#). The AltAI-2 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-2 neural processing unit has a low power consumption.

To execute a spiking neural network, connect the backplane with the AltAI-2 neuromorphic processing unit to the computer via a [PCI - E](#) or [USB](#) interface.

The main structural unit of the AltAI-2 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 behavior of integrator neurons with a leak (Leaky Integrate-and-Fire Neuron, or LIF neurons).

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-2 neural processing unit 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-2 neuromorphic processor provides the ability to scale by increasing the number of VLSI chips. For questions regarding scaling, please contact Kaspersky specialists.

[General information about the AltAI-2 neuromorphic processor](#)

The table below provides general information about the AltAI-2 neuromorphic processor and its components:

General information about the AltAI-2 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	8,192
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

Kaspersky Neuromorphic Platform backends

The platform provides several independent backends for execution of [spiking neural networks](#) on various computing devices.

Kaspersky Neuromorphic Platform backends run spiking neural networks on various [computers](#), perform static verifications of the spiking neural network design, and facilitate data exchange between the neural network and the [external environment](#).

Kaspersky Neuromorphic Platform supports the following types of backends:

- Backends for running spiking neural networks: single-thread and multi-thread CPU backends, and AltAI SNN backend.
- Backends for training and execution of artificial neural networks converted to spiking neural networks: AltAI ANN2SNN backends.

If necessary, you can implement and use your own backends.

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

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](#).
- *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++](#).

Backends for execution of spiking neural networks

Kaspersky Neuromorphic Platform supports the following types of backends for execution of spiking neural networks:

- Single-thread and multi-thread [CPU backends](#)
- Backend for execution of spiking neural networks on an [AltAI-2 neuromorphic processor](#) (AltAI SNN backend)

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.

The `SingleThreadedCPUBackend` currently supports [neurons based on](#) AltAILIF, BLIFATNeuron, and SynapticResourceSTDPBLIFATNeuron models and [synapses based on](#) DeltaSynapse, AdditiveSTDPDeltaSynapse, and SynapticResourceSTDPDeltaSynapse models.

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.

The `MultiThreadedCPUBackend` currently supports neurons based on `BLIFATNeuron` and `SynapticResourceSTDPBLIFATNeuron` models and synapses based on `DeltaSynapse` and `SynapticResourceSTDPDeltaSynapse` models.

Backend for GPUs with CUDA support

The backend performs computations in parallel on multiple GPU cores, thereby speeding up the execution and learning process for a [spiking neural network](#) ^[2]. The interface to the backend is implemented in the `CUDABackend` class.

You can use `CUDABackend` to train and execute neural networks on devices with NVIDIA GPUs that support CUDA technology.

`CUDABackend` currently supports neurons based on the `BLIFATNeuron` model and synapses based on the `DeltaSynapse` model.

AltAI SNN backend

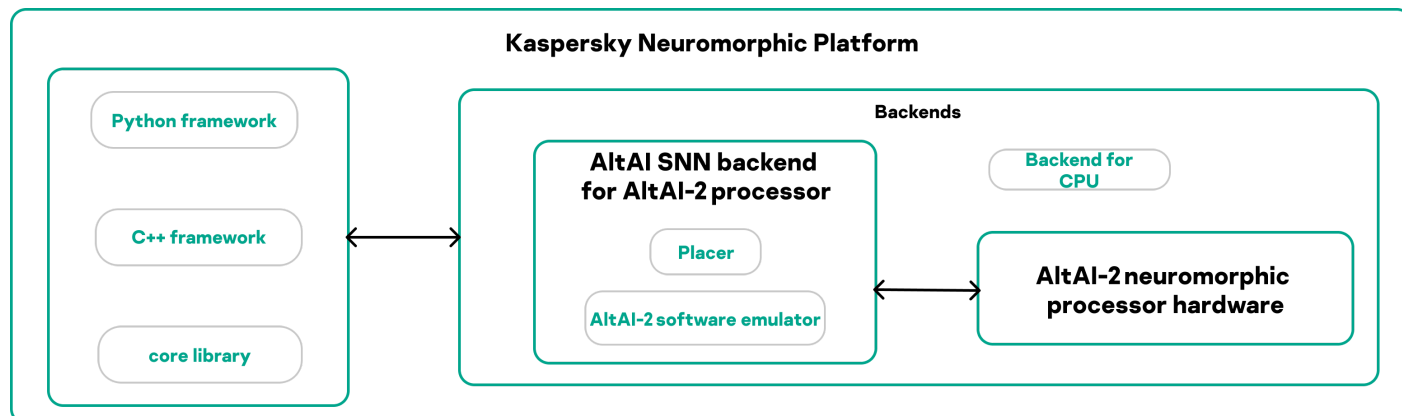
You can use an AltAI SNN backend for execution of spiking neural networks on an [AltAI-2 neuromorphic processor](#). The interface to the backend is implemented in the `AltAIBackend` class. The `AltAIBackend` class provides the standard methods of the `Backend` base class.

You can select the hardware processor of the AltAI-2 neuromorphic processor or a software emulator as the device type. The hardware processor unit uses a fixed core configuration. When you select a software emulator as the device type, you can [change its configuration settings](#).

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-2](#) 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-2 hardware processor.



Scheme of interaction between the platform and the AltAI SNN backend

Backends for training and execution of artificial neural networks

Kaspersky Neuromorphic Platform provides the following backends for training and execution of [artificial neural networks](#) that were converted into spiking neural networks (hereinafter also referred to as "AltAI ANN2SNN backends"):

- *AltAI-2 neuromorphic processor hardware backend*
- *AltAI-2 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-2 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.

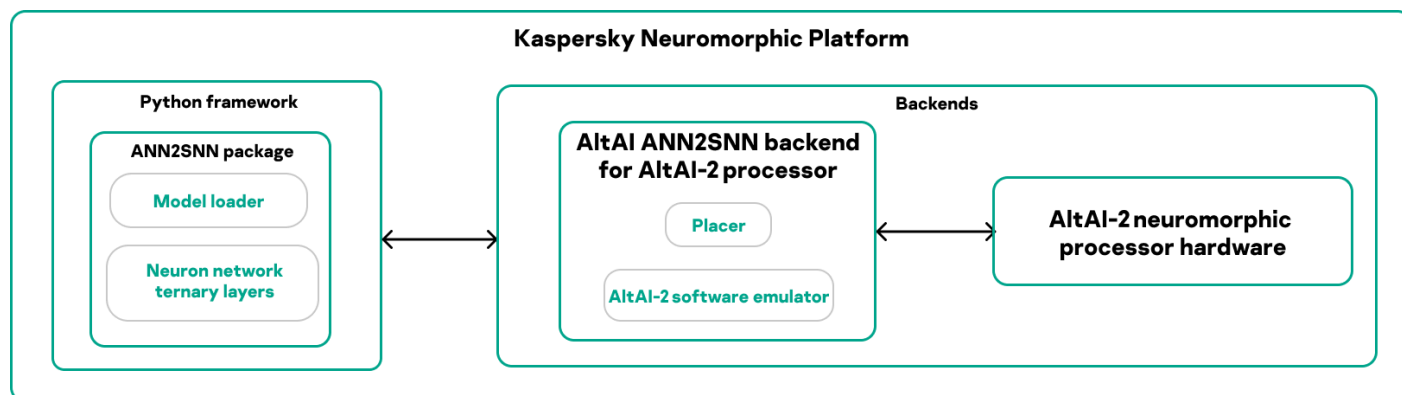


Diagram of interaction between platform components when using the AltAI ANN2SNN package

Neural network description format

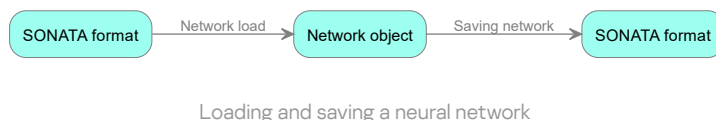
Kaspersky Neuromorphic Platform uses the [SONATA](#) format to describe the topology of the neural network.

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.



To view the structures of files describing groups of neurons and synapses, please refer to the [Appendix](#).

Kaspersky Neuromorphic Platform provides functions for loading and saving a neural network in the `SONATA` format. The functions are implemented in the `sonata` namespace, which is provided in both the [C++ framework](#) and the [Python framework](#).

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' UUIDs 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

To view the structure and an example of a basic format file, please refer to the [Appendix](#).

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

The Kaspersky Neuromorphic Platform C++ framework provides the [native](#) namespace, which implements functions for loading [SpikeMessages](#) from basic format files and saving them to basic format files.

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 [IndexConverter](#) data converter for further processing in the input channels.

Kaspersky Neuromorphic Platform architecture

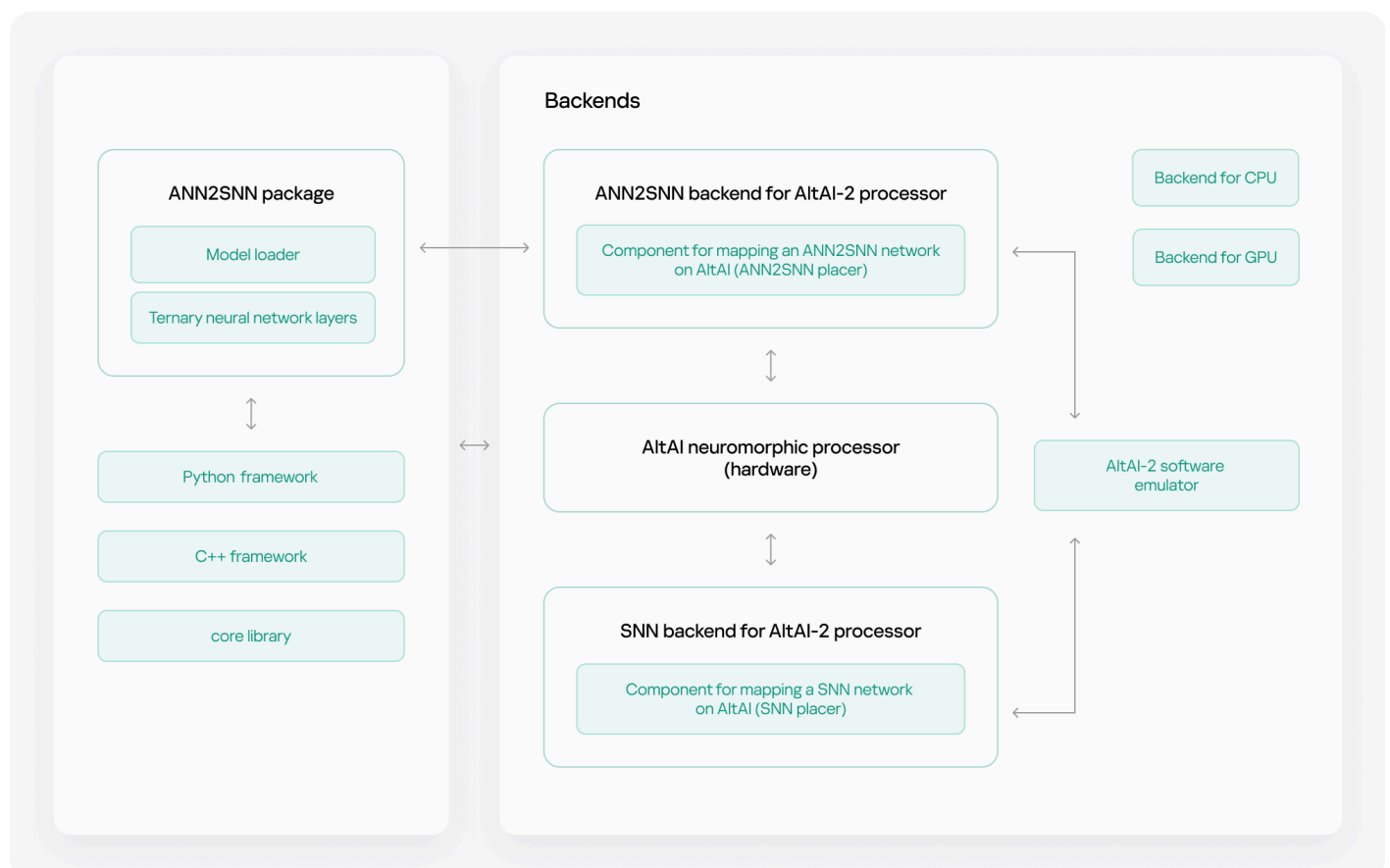
Kaspersky Neuromorphic Platform includes the following components:

- [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-2 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-2 neuromorphic processing unit](#). Resulting neural networks are adapted for execution on the AltAI-2 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-2 neuromorphic processing unit, or a software emulator of the latter.
- The [core](#) library. Provides the classes for interacting with backends and performing application tasks.
- [Backends](#). Start training and execution of a spiking neural network on various computing devices (for example, neuromorphic processor AltAI-2). 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.



Changing the configuration settings of the software emulator

If you select Software Emulator as the device type when using the [AltAI SNN](#) backend, you can change the configuration settings of the [AltAI-2 neuromorphic processor](#) cores.

To change the configuration settings of the software emulator:

1. Change the configuration settings of the AltAI-1 neuromorphic processor cores by using the `load_core_grid_params()` method of the software emulator.

Call `load_core_grid_params` method before initializing the backend.

2. Change the location of the input and output ports by using the `make_standart_north_port()` function of the software emulator.

General operating principles of Kaspersky Neuromorphic Platform

This section contains a description of the general operating principles of Kaspersky Neuromorphic Platform.

Data exchange 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 other platform objects and the external objects.

Data exchange between the backend and the external environment

Data transfer between the backend and the external environment is carried out via messages containing spikes. On the backend, spike messages accept input projections. Spike messages are sent to the external environment via subscriptions.

Spike messages can be transmitted synchronously (by user request) or by backend signal. To receive spikes in synchronous mode, the method that blocks the data source is called.

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. Input channels are used to convert received data into messages containing spikes.

A typical loop of data receipt from the external environment by a neural network includes the following steps:

1. According to the specified function, the `IndexConverter` [data converter](#) or [SequenceConverter](#) converts data received from the external environment into [SpikeMessages](#).
2. The [InputChannel1](#) receives converted messages from the data converter.
3. The `MessageEndpoint` associated with the `InputChannel1` sends spikes to the input projection of the neural [Network](#).

The neural network transmits spikes converted into data to the external environment via output channels connected to the specified populations. A typical loop of data transfer from a neural network to the external environment includes the following steps:

1. The `MessageEndpoint` subscribes to the [messages](#) of a sender.
2. The `MessageEndpoint` sends messages from the associated population.
3. The `MessageBus` delivers messages from the sender to the `MessageEndpoint` associated with the [OutputChannel](#).
4. The `MessageEndpoint` associated with the `OutputChannel` receives messages.
5. The `OutputChannel` receives messages and converts them into the required data format.

If you have loaded a [model](#) storing a neural network and input and output channels into the `ModelExecutor`, the sent messages are automatically transmitted to the message endpoint via the message bus when the model is executed on the backend.

Transmission of data, such as the membrane potential, is also possible through a [monitoring subsystem](#).

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

Data exchange between populations and projections

Populations and projections share data via [messages](#). To exchange messages, populations and projections must be pre-loaded on the [backend](#).

Operation of the [backend for the CPU](#) is based on the exchange of messages between populations and projections.

CPU backend operation

The CPU backend operation is based on the [message exchange](#) between projections and populations. The platform objects can also exchange messages with the objects of the [external environment](#) [\[2\]](#). The backend provides a `MessageBus` for message exchange.

The typical backend work loop includes the following steps:

1. During initialization of the backend, the loaded synapse projections are subscribed to messages from associated populations, and the loaded populations are subscribed to messages from associated projections. If necessary, you can manually subscribe populations and projections to each other by using the `Backend::subscribe()` [method](#) or `MessageEndpoint::subscribe()`.
2. The `InputChannel` input channels send data to the `MessageBus`.
3. The backend computes the function for [neurons](#) [\[2\]](#) of populations.
4. Neurons of populations generate [spikes](#) [\[2\]](#). The conditions for generating spikes are determined by the [types of neurons](#) in the population and the values of their attributes.

5. Populations of neurons send messages (for example, spike messages) to the `MessageBus` via `MessageEndpoints`.
More than one projection can be subscribed to the messages from the population.
6. As defined by the subscriptions, the `MessageBus` sends data received from populations to [projections](#) or to the [OutputChannel](#) output channels. More than one projection can be subscribed to the messages containing spikes.
7. The backend computes [synaptic impacts](#).
8. Synapse projections send synaptic impacts with the computation results to the `MessageBus` using `MessageEndpoints`.
9. The `MessageBus` sends data received from the projections to the associated neuron populations.

Processing messages by subscription

A [subscription](#) provides for filtering of [messages](#). Messages are filtered according to the unique identifiers of senders and recipients defined in the subscription object. Messages are stored in a subscription until they are requested by the recipient or until they are removed from the subscription.

The typical subscription message handling loop includes the following steps:

1. Send messages via message endpoints.

Platform objects send messages by using the `MessageEndpoint::send_message()` method.

2. Route and filter messages.

After calling the `route_messages()` method, the `MessageBus` distributes messages to subscriptions according to the unique identifiers of the message senders and receivers that are defined in them.

3. Deliver messages to recipients.

The recipient requests messages by using the `receive_all_messages()` and [unload_messages\(\)](#) methods of the `MessageEndpoint` class.

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 computer must have software installed as defined in the [hardware and software requirements](#).

The deb packages are installed on Linux.

The following packages must be installed for application development:

- Packages from the `pkg-deb.zip` archive:
 - `knp-cpu-single-threaded-backend_<knp_version>_amd64.deb`: a deb package with a CPU single-threaded backend dynamic library.
 - `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`: a deb package with a CPU multi-threaded backend dynamic library.
 - `knp-gpu-cuda-backend_<knp_version>_amd64.deb`: a deb-package with a backend library for GPUs with support for CUDA technology.
 - `knp-cpp-framework_<knp_version>_amd64.deb`: a deb package with a C++ framework dynamic library.
 - `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-gpu-cuda-backend-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-gpu-cuda-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 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 containing the API reference.
- `knp-altai-backend_<motiv_version>_amd64.deb`: a deb package that contains an [AltAI SNN backend](#) dynamic library.
- `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 with the ANN2SNN package, [AltAI ANN2SNN backend](#) dynamic library for the [AltAI-2 neuromorphic processor](#), and a component for placing the neural network on AltAI-2.

where:

- `<knp_version>` is the Kaspersky Neuromorphic Platform version.
- `<motiv_version>` is the version of the backends for the AltAI-2 neuromorphic processor and the ANN2SNN package.

To install the deb packages:

1. Download the `pkg-deb.zip` archive and the packages from the [platform repository](#).
2. Unpack the `pkg-deb.zip` archive.

3. To install deb packages containing CPU backend dynamic libraries, a GPU backend, and an AltAI SNN backend, run the following commands:

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

When installing the knp-altai-backend_<motiv_version>_amd64.deb package, accept the AltAI-2 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 C++ framework dynamic library, run the following command:

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

5. To install dev-deb packages, run the following commands:

```
sudo dpkg -i <path-to-deb-package>/knp-cpu-multi-threaded-backend-dev_<knp_version>_amd64.deb
sudo dpkg -i <path-to-deb-package>/knp-cpu-single-threaded-backend-dev_<knp_version>_amd64.deb
sudo dpkg -i <path-to-deb-package>/knp-gpu-cuda-backend-dev_<knp_version>_amd64.deb
sudo dpkg -i <path-to-deb-package>/knp-altai-backend-dev_<motiv_version>_amd64.deb
sudo 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-2 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 a deb package containing a Python framework for an AltAI SNN backend, run the following command:

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

7. To install a deb package containing an ANN2SNN package and an AltAI ANN2SNN backend dynamic library, run the following command:

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

When installing the knp_ann2snn.deb package, accept the AltAI-2 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.

8. To install the deb package with platform usage examples, run the following command:

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

9. To install the deb package with the API reference guide, run the following command:

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

Installing Python development packages

The computer must have software installed as defined in the [hardware and software requirements](#).

The packages are installed on Linux.

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](#) from the `pkg-deb.zip` archive.
 - `knp_ann2snn.deb`: a deb package with the ANN2SNN package, [AltAI ANN2SNN backend](#) dynamic library for the [AltAI-2 neuromorphic processor](#), and a component for placing the neural network on AltAI-2.
- whl packages:
 - `knp-<knp_version>-py3-none-any.whl`: a whl package containing a Python framework for an AltAI SNN backend from the `pkg-deb.zip` archive.
 - `knp_ann2snn_x86_64-<motiv_version>-py3-none-any.whl`: a whl package with an ANN2SNN package, AltAI ANN2SNN backend dynamic library for the [AltAI-2 neuromorphic processor](#), and a component for placing the neural network on AltAI-2.

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 dynamic library for the AltAI-2 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-2 neuromorphic processor and the ANN2SNN package.

To install deb packages for Python development:

1. Download the `pkg-deb.zip` archive and the `knp_ann2snn.deb` package from the [platform repository](#).
2. Unpack the archive.

3. To install a deb package containing a Python framework for an AltAI SNN backend, run the following command:

```
sudo 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.
4. To install a deb package containing an ANN2SNN package and an AltAI ANN2SNN backend dynamic library, run the following command:

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

When installing the knp_ann2snn.deb package, accept the AltAI-2 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 the whl packages.

To install whl packages:

1. Download the pkg-python.zip archive and the whl packages from the [platform repository](#).
2. Unpack the archive.
3. 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.

4. To install a whl package containing an ANN2SNN package and an AltAI ANN2SNN backend dynamic library, do one of the following:

- If you need to install a package on an x86-based computer, run:

```
pip3 install --break-system-packages knp_ann2snn_x86_64-<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-2 neuromorphic processor and the ANN2SNN package.

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 backend dynamic library, GPU backend, and an `AltAI SNN` backend, run the following commands:

```
sudo apt purge knp-cpu-single-threaded-backend
sudo apt purge knp-cpu-multi-threaded-backend
sudo apt purge knp-gpu-cuda-backend
sudo apt purge knp-altai-backend
```

2. To uninstall the deb package with a C++ framework dynamic library, run the following command:

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

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

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

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

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

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

```
sudo apt purge knp_ann2snn
```

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

```
sudo apt purge knp-examples
```

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

```
sudo 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 containing a Python framework for an `AltAI SNN` backend, run the following command:

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

2. To remove a deb package containing an `ANN2SNN` package and an `AltAI ANN2SNN` backend dynamic library, run the following command:

```
sudo 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 --break-system-packages 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 an `AltAI ANN2SNN` backend dynamic library, do one of the following:

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

```
pip3 uninstall --break-system-packages knp_ann2snn_x86_64-<motiv_version>-py3-none-any.whl
```

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

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

where `<motiv_version>` is the version of the backends for the AltAI-2 neuromorphic processor and the `ANN2SNN` package.

Removing the platform source code

To remove the platform source code,

delete the local platform repository and the directory to which the platform source code archive [was unpacked](#).

Scenario for building a platform project or an application solution in C++

You can build a platform or an application solution in C++ from source code. To build an application solution, you can add the platform as a dependency using the `add_subdirectory` function in CMake or using a dependency manager (for example, CPM.cmake).

When developing application solutions in Linux, you can [install the required deb packages](#) instead of building the solution.

The computer must have software installed as defined in the [hardware and software requirements](#).

You can build for Linux in a Docker container. To do so, you can use the `knp-build-image` Docker image that comes with the [platform distribution kit](#).

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

1 Preparing for the build

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

- [Clone the platform repository](#).
- [Download the archive with the source code](#) from the platform repository and unpack it to the working directory.

After receiving the platform source code, install the necessary version of the [Boost](#) library.

When working in Windows, it is recommended to install the library [precompiled](#) for Visual Studio. For example, for Visual Studio 2022, the compiler version is 14.3. Accordingly, to install the Boost library compiled for Visual Studio, you must download an executable file named according to the following format: `boost_<Boost version>-msvc-14.3-<Windows OS bit rate>.exe`.

2 Platform build configuration

In Windows, for CMake, you must specify the directory in which the Boost library is installed. You can do this in one of the following ways:

- Specify the path to the installed Boost library in the `Boost_ROOT` setting in the `CMakePresets.json` file located in the root directory containing the platform source code.

CMake caches the values of variables. If CMake cannot find the Boost library after you define the `Boost_ROOT` setting, you must [regenerate the CMake cache](#).

- Create the `Boost_ROOT` environment variable.

- Define the path to the installed Boost library as a CMake invocation parameter. To do so, use the following parameter when configuring the build:

```
-DBOOST_ROOT = <path to installed Boost library>
```

If you are building a platform with a Python framework, review the [build specifics](#).

If you are building a platform for Linux, configure the build using the CMake build system. For example, you can use the following command:

```
cmake -S. -Bbuild --preset <build preset> -DCMAKE_C_COMPILER = <compiler> -DCMAKE_CXX_COMPILER = <compiler>
```

where:

- `<build preset>` is the name of the preset file that describes the build parameters.

You can specify one of the presets described in the `CMakePresets.json` file located in the root directory with the platform source code. For a description of the presets provided by the Kaspersky Neuromorphic Platform, see [Appendix](#).

You are not required to use the `--preset <build preset>` option when performing the build configuration.

- `<compiler>`: compilers for C and C++.

When configuring the build, you can set the [platform configuration settings](#).

For more details on configuring a build using CMake, refer to the [CMake documentation](#).

If you are building the platform in Windows, [configure the build](#) in Visual Studio. To do so, open the project by selecting the required `CMakeLists.txt` file and configure the cache. To build the platform, you must select the `CMakeLists.txt` file located in the root directory containing the platform source code.

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

3 Starting the platform 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.

To start the build on Linux, you can use the command:

```
cmake -B build.
```

To start a build in Windows in Visual Studio, on the **Build** tab, select **Build all**.

4 Building an application solution

To build an application solution, create a `CMakeLists.txt` file and [describe the build process in it](#). Add the platform as a dependency by specifying the platform directory using the `add_subdirectory` function. Specify the executable file by using the `add_executable` function. Use the `target_link_libraries` function to add the platform libraries that need to be linked to the project. You can use the following libraries:

- [neuron_traits](#)

- [synapse_traits](#)
- [core](#)
- [meta](#)
- [backends](#)
- [framework](#)

Configure the build for the application solution. If building in [Windows](#), select the [CMakeLists.txt](#) file of the application solution when opening the project. Start building the application solution.

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.

Specifics of building the platform with a Python framework

Building a platform with the Python framework has the following considerations:

- Working with the Python framework requires Python 3.12 or higher and a Boost library whose version corresponds to the Python version.
- If necessary, you can use a virtual environment to work with the Python framework. To create a virtual environment, go to the working directory and execute the following command:

```
python3 -m venv venv
```

To connect to the created virtual environment, execute the following command:

```
source venv/bin/activate
```

- The platform is automatically built with the Python framework if the device has stubgen installed on it. If stubgen is not available on the device, you can do one of the following:

- Install the mypy package to access stubgen. To do this, run the following command:

```
pip install mypy
```

- Enable building of the platform with the Python framework via the [KNP_PYTHON_FRAMEWORK_BUILD](#) build configuration setting.
- If the device has stubgen installed, a whl package with a Python framework is automatically built when the build starts. If stubgen is not available on the device, you can use the build configuration setting [KNP_PYTHON_BUILD_WHEEL](#) to enable building a whl package.

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** or **Configure knp**.

CMake will begin working on the build configuration.

If network errors occur, please run the build configuration again.

After CMake finishes its configuration work, start the build.

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](#), and [framework](#).

Before building a project in Windows, [define the](#) `Boost_ROOT` setting.

For more details on configuring a build using CMake, refer to the [CMake documentation](#).

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.

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
- Neuron based on the `BLIFATNeuron` model
- Neuron based on the `SynapticResourceSTDPBLIFATNeuron` model

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 `AdditiveSTDPDeltaSynapse` model
- Synapse based on the `DeltaSynapse` model
- Synapse based on the `SynapticResourceSTDPDeltaSynapse` model

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
<code>messaging</code>	Namespace that contains message interfaces.
<code>tags</code>	A namespace that contains the names and value types of known tags.
<code>Backend</code>	Base class for specific backend implementations.
<code>BaseData</code>	Structure defining a set of base data.
<code>continuously_uid_generator</code>	Class implementing a generator of unique IDs.
<code>Device</code>	Base class of devices.
<code>MessageBus</code>	Class implementing an interface for the message bus.
<code>MessageEndpoint</code>	Class implementing an interface for the message bus endpoint.
<code>Population</code>	Template class that implements the container of <code>neurons</code> of the same model.
<code>Projection</code>	Template class that implements the container of <code>synapses</code> of the same type.
<code>Subscription</code>	Class implementing the message subscription.
<code>TagMap</code>	Class implementing an interface for the tag dictionary.
<code>UID</code>	Class defining unique identifiers.
<code>uid_hash(.)</code>	Function implementing hashing of UIDs.

The figure below shows how the library components interact.

In Kaspersky Neuromorphic Platform, the `BaseData` contains basic information about an object and includes the unique identifier (UID) and an associative array of `TagMap` tags. The `BaseData` 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.

`Populations` and `Projections` store data that the backend processes and passes to other platform objects via messages implemented in classes of the `messaging` namespace. Messages contain the sender's UID in the header.

A Backend contains a Device and a MessageBus. To connect to the MessageBus, platform objects create MessageEndpoints that are used by the message bus to interact with these objects during the message exchange process. Each MessageEndpoint has a Subscription container.

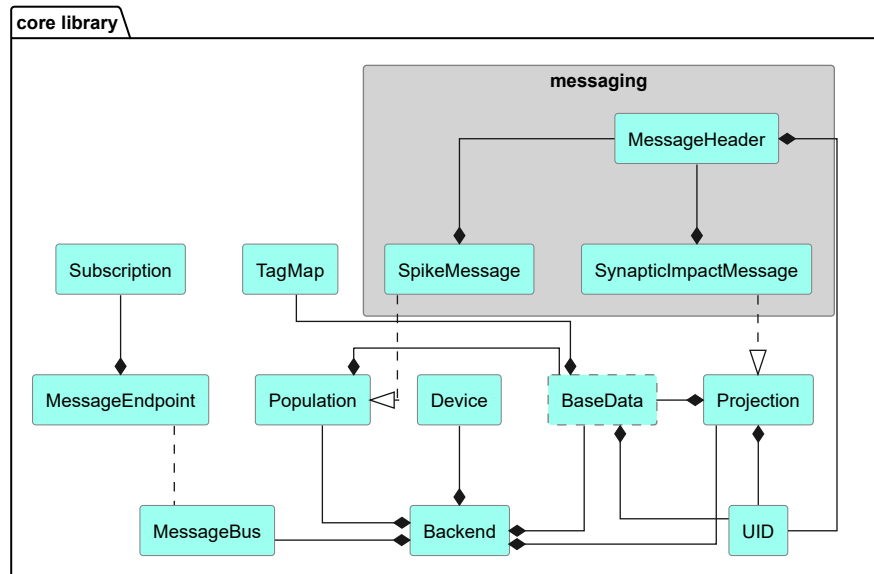


Diagram of interactions between components in the core library

messaging namespace

The `messaging` namespace implements interfaces for accessing messages. 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.

Message exchange is provided by the `MessageBus` message bus. 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 type	Description	Message structure
SpikeMessage	A message to a synapse projection containing a spike [?]. More than one projection can be subscribed to the messages containing spikes.	<code>header_</code>—message header containing the UID of the input channel or population whose neurons generated the spikes and the time when the spikes were generated. <code>neuron_indexes_</code>—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 [?].	<code>header_</code>—message header containing the projection UID and the time when the message was sent. <code>postsynaptic_population_uid_</code>—UID of the postsynaptic population. <code>Presynaptic_population_uid_</code>—UID of the presynaptic population. <code>is_forcing_</code>—flag to enable or disable plasticity in a synapse projection. If the synapse projection uses synaptic plasticity, the attribute is <code>true</code>, otherwise it is <code>false</code>. <code>impacts_</code>—vector containing the following attributes: <code>connection_index_</code>—connection index. <code>impact_value_</code>—synaptic impact value. <code>synapse_type_</code>—type of synapse. <code>presynaptic_neuron_index_</code>—index of the presynaptic neuron from which the spike was received by the projection synapse. The neuron index is unique within the population. <code>postsynaptic_neuron_index_</code>—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 <code>impacts_</code> vector may contain more than one set of the listed attributes to convey synaptic impacts to multiple neurons in the same population.

The `messaging` namespace provides functions that let you receive messages from an input data stream, send messages to an output data stream, and compare messages with each other.

Kaspersky Neuromorphic Platform provides methods and functions for interacting with messages. For example, you can use the `MessageEndpoint::send_message()` method to send a specific message through a message endpoint to a message bus, and you can use the functions of the `storage` namespace to load and save `SpikeMessages` in the [basic KNP format](#).

For more information about the `messaging` namespace functions and platform object methods, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

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.

Backend class

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

- Interact with physical devices (for example, receive information about power consumption).
- Manage 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.

Prior to initializing the backend, you can download projections containing [synapses](#) and populations containing [neurons](#). To do so, you can use the [load_all_projections\(\)](#) and [load_all_populations\(\)](#) methods of the `Backend` base class or the corresponding methods of a specific backend class. If you have previously created a [Model](#) to which a neural [Network](#) storing [Populations](#) and [Projections](#) has been passed, you can use the [ModelLoader::load\(\)](#) method to load it into the backend.

Specific classes of backends are inheritors of the `Backend` base class. To initialize a specific backend, you must redefine its methods in the inherited class. During initialization, 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. If necessary, you can subscribe the required platform objects to messages by using the [subscribe\(\)](#) method.

To select a device for executing the neural network, first call the [get_devices\(\)](#) method that provides the specific type of computers. The computer type is an inheritor of the [Device](#) class. Inheritors are defined in the [devices](#) namespace and implement specific devices. Each device has its own set of functions and properties. Next, you need to call the [select_devices\(\)](#) or [select_device\(\)](#) method to select a specific device from those provided while passing unique device identifiers or pointers for `Device` class instances to the method.

The `Backend` base class also provides methods that you can use to do the following:

- Start and stop neural network training.
- Enable or disable the use of synaptic plasticity during neural network training.

- Start and stop neural network execution.
- Receive the following data:
 - [Basic backend data](#).
 - [Message bus](#). Using the message bus, the backend ensures the message exchange between the platform objects, as well as sending messages from [modifiers](#) to [neurons](#) and [synapses](#).
 - Message bus [endpoint](#).
 - Neural network execution status on the backend.
 - Devices supported by the backend.
 - Neural network execution step number.

For more information about methods of the `Backend` base class, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

BaseData structure

The `BaseData` structure defines a set of base data that is common to some platform objects. For example, `BaseData` is used in objects of the `Backend`, `Device`, `Population`, and `Projection` classes.

`BaseData` includes the following attributes:

`uid`

Unique identifier of the object.

`tags`

Associative `TagMap` array consisting of the tags used by the object and their values.

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.

For more information about methods of the `continuously_uid_generator` class, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

Device class

The `Device` class is the base class for the device classes. The class inheritors override methods 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.

Each object of the `Device` class contains [basic data](#) for a specific device.

For more information about the `Device` base class and its methods, please refer to the [API documentation of Kaspersky Neuromorphic Platform](#).

MessageBus class

The `MessageBus` class implements an interface for accessing a message bus. The message bus provides for the receipt, distribution and delivery of [messages](#) between objects connected to it.

The `MessageBus` class provides methods that let you do the following:

- Create message buses.
- Create message bus endpoints.
- Receive, distribute and deliver messages between objects connected to a message bus.
- Get the message endpoints being used.

A message bus provides a set of [MessageEndpoints](#) through which platform objects and [external environment](#) objects can subscribe to messages or send messages. You can create message endpoints for connecting to the message bus and then subscribe the created message endpoints to certain types of messages using the [MessageEndpoint::subscribe\(\)](#) subscribe method. 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.

For more information about methods of the `MessageBus`, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

MessageEndpoint class

The `MessageEndpoint` class implements an interface for accessing a message endpoint that subscribes to the [messages](#) and sends them using the [MessageBus](#).

You can create a message endpoint in the following ways:

- Use the [MessageBus::create_endpoint\(\)](#) method.
- Use the [MessageEndpoint](#) transfer constructor.

A message endpoint can store a set of [Subscriptions](#) to messages from various senders and for various recipients, and can store a set of UUIDs of senders to which it is subscribed. Subscriptions are used to filter and store messages received by the message bus.

The `MessageEndpoint` class provides methods that let you do the following:

- Subscribe a message endpoint to messages of a specific type.
- Unsubscribe from messages from specific senders.
If you delete a message endpoint, the associated subscription objects are also deleted. Message transmission through this message endpoint stops.
- Receive messages from a sender to which a message endpoint is subscribed.
Before calling the method, you must use the [MessageBus::route_messages\(\)](#) method for the message endpoint to receive messages.
- Send messages via a message bus.
- Receive subscriptions associated with a message endpoint.
- Receive a set of UUIDs of senders to which a message endpoint is subscribed.
- Receive a unique identifier for the recipient of messages by subscription.

For more information about methods of the `MessageEndpoint`, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

Population class

The `Population` template class implements a container that contains [neurons](#) belonging to the same model. When creating a population object, you must specify the number of neurons and the generator of the specific type of neurons. You can specify one of the neuron types supported by the [neuron_traits](#) library.

You can pass population data to the backend and calculate their attributes by using backend methods. To pass population data to the backend, you can use the `Backend::load_all_populations()` method. After computation, the new population attributes are stored in the backend. To get the current neural network parameters from the backend, you can call 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.

The `Population` class provides methods that let you do the following:

- Add neurons to a previously created population.
- Remove one or more neurons from a population.
- Get basic population data.
- Get attributes of the population neurons.
- Get information about the number of neurons in a population.

For more details about the methods of `populations` and `backends`, please refer to the API documentation of Kaspersky Neuromorphic Platform.

Projection class

The `Projection` template class implements a container that contains `synapses` of one of the types supported by the `synapse_traits` library. When creating a projection object, you must specify the unique identifiers of the presynaptic and postsynaptic populations that need to be linked by this projection.

A projection is a set of neuron connections of the presynaptic and postsynaptic `Populations`. Neuron populations and synapse projections `share data` via `messages`. A synapse projection receives `SpikeMessages` containing `spikes` from the presynaptic population and sends `SynapticImpactMessages` containing `synaptic impacts` to the postsynaptic population.

You can use the `Backend::load_all_projections()` projection load method to load the created projection data into the backend to calculate its attributes.

The `Projection` class also provides methods that let you do the following:

- Add synapses to a projection.
- Remove synapses from a projection. You can remove synapses according to some specified criterion, or you can remove synapses associated with a specific presynaptic or postsynaptic neuron.
- Get basic projection data.
- Get unique identifiers of populations associated with a projection.
- Get attributes of projection synapses.
- Get information about the number of projection synapses.
- Lock and unlock the capability to change the weights of synapses.

Each synapse projection has the following attributes:

- 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.
- Attributes used by synapses of the same type.
- 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`.

For more information about methods of the projection, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

Subscription class

The `Subscription` class implements the functionality for subscribing to messages and stores the received [messages](#). You can create a subscription in one of the following ways:

- Use the [Subscription](#) constructor.
- Use the [MessageEndpoint::subscribe\(\)](#) method.

A subscription provides for [filtering of messages](#). Messages are filtered according to the unique identifiers of senders and recipients defined in the subscription object. Messages are stored in a subscription until they are requested by the recipient or until they are removed from the subscription.

Subscriptions are stored at message endpoints to a message bus. If the [MessageEndpoint](#) object that stores the subscription object is deleted, the subscription object is also deleted. Messages defined by this subscription stop being transmitted.

The `Subscription` class provides methods that let you do the following:

- Add message senders to a subscription.
- Remove message senders from a subscription.
- Add messages to a subscription.
- Delete messages that are stored in a subscription.

- Get unique identifiers of message senders and recipients.
- Receive messages stored in a subscription.
- Check whether a subscription is associated with a sender that has a specific unique identifier.

For more information about methods of the `Subscription` class, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

TagMap class

`TagMap` class 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. An associative array of tags is a component of the `BaseData` of platform objects.

The `TagMap` class provides methods that let you do the following:

- Get the value of a tag by passing its name to the method.
- Check for records in the associative array.

For more information about methods of the `TagMap` class, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

UID class

The `UID` class implements a unique identifier for an object. The identifier is created by using the `UID` constructor and is part of the `BaseData` of platform objects.

The `UID` class provides operators that you can use to compare the identifiers of objects and verify that the identifiers are correct. If necessary, you can get the hash value of an object identifier by using the `uid_hash()` function.

For more information about constructors and operators of the `UID` class, please refer to the [API documentation of Kaspersky Neuromorphic Platform](#).

uid_hash() function

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

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 namespace

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

The functions and classes implemented in the namespace are used in backends to get information about devices. To select a device for executing the neural network, first call the `Backend::get_devices()` method that provides computer types. Computer types are inheritors of the `Device` class. Inheritors are defined in the `devices` namespace and implement specific devices. Each device has its own set of functions and properties. To select a specific device from those provided, you must call the `select_devices()` or `select_device()` method of the `Backend` base class while passing to it the UIDs of devices or pointers to the `Device` class 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.

If necessary, you can implement and use your own classes for interaction with devices when [working with source code](#). To do this, implement a device class within the backend, provide an interface for the device, and implement a device loading method that returns a pointer to an instance of the `Device` class.

The Kaspersky Neuromorphic Platform currently has interfaces for the CPU, GPU with CUDA support, and the AltAI-2 neuromorphic processing unit. When using an AltAI-2 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. When you select a software emulator as the device type, you can [change its configuration settings](#).

For more information about the `devices` namespace functions, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

C++ framework

C++ framework is designed to simplify the processes of creating [spiking neural networks](#) and training them. Resulting neural networks can be executed on a central processor, an [AltAI-2 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.
- 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
<code>coordinates</code>		A namespace that contains the spatial coordinates of the neuron.
<code>data_processing</code>		A namespace that provides the functionality for data processing and training of classification models, and for processing images for classification tasks.
<code>inference_evaluation</code>		A namespace that provides the functionality for evaluating the quality of a classification model and writing the results in a convenient format.
<code>io</code>		A namespace that contains the <code>input</code> , <code>output</code> , and <code>storage</code> namespaces.
	<code>input</code>	A namespace that contains interfaces for accessing the input channel and the converter for data coming from the external environment [?].
	<code>output</code>	A namespace that contains interfaces for accessing the output channels.
	<code>storage</code>	A namespace that contains functions for loading and saving <code>SpikeMessage</code> messages.
<code>modifier</code>		A namespace that contains interfaces to message handlers.
<code>monitoring</code>		A namespace that contains the interface to the observer and model monitoring functions.
<code>normalization</code>		A namespace that contains the functions and functors for normalizing the parameters of synapses and neurons in neural networks.
<code>population</code>		A namespace that contains functions for generating populations of neurons.
<code>projection</code>		A namespace that contains functions for generating synapse projections and generating synapses with default attribute values.
<code>sonata</code>		A namespace that contains functions to load and save the neural network in the <code>SONATA</code> format.
<code>synchronization</code>		A namespace that contains a function for getting the current state of the neural network from the backend.
<code>BackendLoader</code>		A class that implements a dynamic library loader.
<code>Model</code>		A class that implements the model.
<code>ModelExecutor</code>		A class that implements the model executor.
<code>ModelLoader</code>		A class that implements the model loader.
<code>Network</code>		A class that implements a neural network object.

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

In Kaspersky Neuromorphic Platform, a neural `Network` can be [loaded](#) in `SONATA` format or it can be manually created. A created neural network can be populated with [Projections](#) of synapses and [Populations](#) of neurons. You can generate projections and populations by using functions of the `projection` and `population` namespaces, respectively. If necessary, you can normalize the attribute values of the neurons of populations and synapses of projections by using functions of the `normalization` namespace.

You can draw a neural Network by using a set of arbitrary or user-defined coordinates from the `coordinates` namespace. The state of the neural Network can be updated from the `Backend` with the `synchronization` namespace function.

The neural Network and the `InputChannel` and `OutputChannel` containers implemented in the `input` and `output` namespaces, respectively, are stored in a `Model`. 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](#).

You can execute a `Model` by using the `ModelExecutor::start()` method. The `ModelExecutor` also contains a `MessageObserver` from the `monitoring` namespace, a message handler from the `modifier` namespace, and a `ModelLoader` whose methods can be used to load a model into a specific backend instance. To process model data, you can use the functions of the `data_processing` namespace. To execute the model in the backend instance, a pointer to the backend instance received from the `BackendLoader` is passed to the `ModelExecutor`. After executing a model, you can evaluate the quality of the model by using the functions of the `inference_evaluation` namespace.

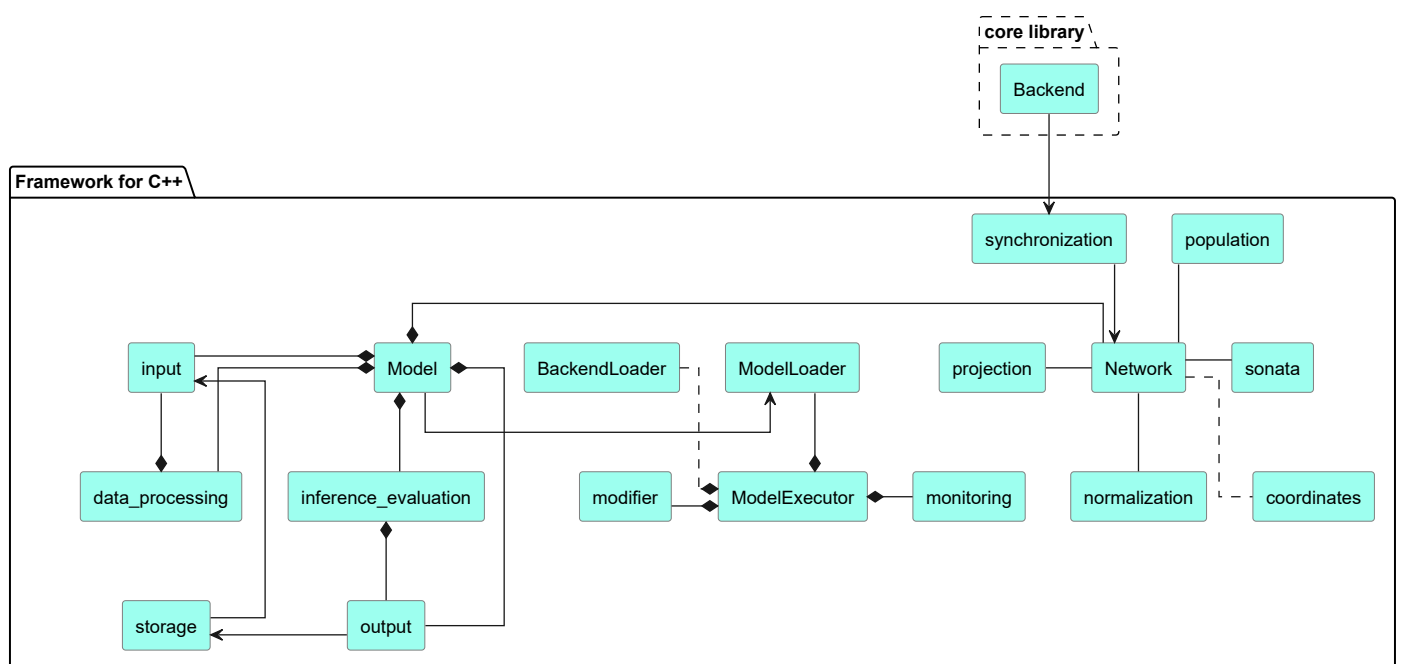


Diagram of interactions between the C++ framework components

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.

For more information about the `coordinates` namespace, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

data_processing namespace

The `data_processing` namespace provides the functionality for data processing for training of classification models, and for processing images for classification tasks.

Dataset base class

The `classification` namespace that is part of the `data_processing` namespace contains the `Dataset` class, which is the base class for the classes of data sets that you can use for training and testing classification models.

The `Dataset` base class provides the following methods:

`split()`

Divides a dataset into training and test samples. The training and test samples consist of vectors of object class label pairs and vectors of binary values indicating the presence or absence of a spike. In turn, the presence of a spike indicates whether the object belongs to a specific class.

`get data for training()`

Receives data for classification model training.

`get data for inference()`

Receives data for classification model inference.

`get steps per frame()`

Receives the number of steps necessary for processing one class of objects.

`get steps required for training()`

Receives the number of steps necessary for classification model training.

`get steps required for inference()`

Receives the number of steps necessary for classification model inference.

`get required training amount()`

Receives information about the number of objects used for classification model training.

`get amount of classes()`

Receives information about the number of classes in the data set.

images namespace

The `classification` namespace also contains the `images` namespace that is used to get information about the classification of images. In turn, the `images` namespace contains the `Dataset` class, which is an inheritor of the `Dataset` base class and provides additional methods for processing images.

The `Dataset` class in the `images` namespace provides the following methods:

`process_labels_and_images()`

Reads a stream of images and their class labels and generates a dataset for image classification model training and inference by converting images to spikes.

`make_training_labels_generator()`

Creates a label set generator corresponding to the image classes for processing by the model during its execution step.

`make_training_images_spikes_generator()`

Creates a generator for a set of images in the form of spikes for processing by the model during its execution step in training.

`make_inference_images_spikes_generator()`

Creates a generator for a set of images in the form of spikes for processing by the model during its execution step in inference.

`make_incrementing_image_to_spikes_converter()`

Creates a converter that converts an image in the form of a vector of bytes into a vector of spikes.

`get_image_size()`

Gets the size of an image, in bytes.

For more information about the `data_processing` namespace classes, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

inference_evaluation namespace

The `inference_evaluation` namespace provides the functionality for evaluating the quality of a model and writing the results in the CSV format.

Functions of the `inference_evaluation` namespace

The `inference_evaluation` namespace implements the following functions to obtain the model evaluation metrics:

`get_precision()`

Retrieves the *precision* metric, which reflects the percentage of positive results that are actually positive.

`get_recall()`

Retrieves the *recall* metric, which reflects the percentage of class objects that were correctly classified by the model out of all objects of this class.

`get_prevalence()`

Retrieves the *prevalence* metric, which reflects the percentage of objects with positive states.

`get_accuracy()`

Retrieves the *accuracy* metric, which reflects the percentage of correctly predicted classes among all predictions.

`get f score()`

Retrieves the *F-score* metric, which reflects the harmonic average between the accuracy and recall.

For formulas to calculate the model quality evaluation metrics, see [Appendix](#).

classification namespace

The `inference_evaluation` namespace contains the `classification` namespace. This namespace implements the `InferenceResult` class, which provides the interface for working with model evaluations based on the classification results for one class of objects. You can use the `InferenceResult` class to receive information about the classification results and the total number of votes. The total number of votes consists of the sum of truly positive, falsely positive, and falsely negative results.

The `classification` namespace also implements the `InferenceResultsProcessor` class, which processes the results of classification model evaluation for all classes of objects. This class implements the following methods:

`process inference results()`

Processes the results of evaluations for all classes of objects.

`get inference results()`

Receives the results of model evaluation for each class of objects.

`write inference results to stream as csv()`

Writes the results of classification model evaluation to the stream in CSV format. A description of the data that is transmitted in CSV format is provided in the [Appendix](#).

For more information about the `inference_evaluation` namespace classes and functions, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

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. When creating a data converter, you must specify the `istream` data stream from which the data will be received, the input data conversion function, and the length of the input data segment. You can change the length of the input data segment if necessary.

The `SequenceConverter` data converter returns a list of indices of the input values that contain spikes as a message of the `SpikeMessage` type. If necessary, you can implement and use custom input data conversion functions.

The `SequenceConverter` class provides methods that you can use to obtain the data stream used by the data converter, and also to call a function to convert data from the `istream` into a `SpikeMessage`.

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.

When creating a data converter, you must specify the `istream` from which the data will be received, and specify the symbol with which the indices of the neurons that generated the spikes will be separated in the `SpikeMessage`.

You can use the methods of the `IndexConverter` class to get the data stream used by the data converter and to call a function to convert data from the `istream` into a `SpikeMessage`.

InputChannel class

The `InputChannel` class implements an input channel that receives messages of the `SpikeMessage` type from the data converter. When creating an input channel, you must specify a unique input channel UID, a `MessageEndpoint`, and a functor for converting data into `SpikeMessages`.

After it is created, the input channel must be associated with a message endpoint and a message receiver by using the `input::connect_input()` method. When a `Subscription` is present, the data converted into `SpikeMessages` can be passed via message endpoints by using methods of the `InputChannel` class.

If you have loaded a `Model` storing a neural `Network` and input and output channels into the `ModelExecutor`, the sent messages are automatically transmitted to the message endpoint via the message bus when the model is executed on the `Backend`.

The `InputChannel` class also provides methods that you can use to get the base data of the input channel.

Each input channel contains the following attributes:

- 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.

For more information about the functions and interfaces of the input namespace, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

modifier namespace

The `modifier_namespace` contains classes that implement modifiers.

`KWtaRandomHandler`

Provides the message handler that receives the vector of `SpikeMessages` and returns a specific number of random indices of neurons that generated spikes.

GroupWtaRandomHandler

This provides a message handler that receives a vector of `SpikeMessages` and breaks them down into groups. In each group, the handler sorts messages based on the quantity of neurons that generated spikes in descending order and returns the indices of neurons that generated spikes in a specific number of sorted messages of the group.

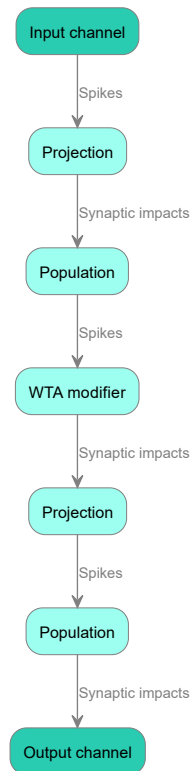
KWtaPerGroup

This provides a message handler that receives a vector of `SpikeMessages` and breaks them down into groups. For each group of messages, the handler returns a specific number of random indices of neurons that generated spikes.

SpikeUnionHandler

Provides message handler that receives the vector of `SpikeMessages` and returns the indices of neurons that generated spikes in at least one message.

The `KWtaRandomHandler`, `GroupWtaRandomHandler`, and `KWtaPerGroup` classes implement the [Winner Takes ALL \(WTA\) mechanism](#). Each of these modifiers receives messages containing spikes (`SpikeMessages`) and sends messages containing synaptic impacts (`SynapticImpactMessages`). Therefore, when building a neural network, the neural layer containing the WTA modifier must be connected to the neuron population at the input and to the synapse projection at the output (see the figure below).



Neural network structure with a WTA modifier

You can add a message handler to the model executor by using the `ModelExecutor::add_observer()` method or the `projection::add_wta_handlers()` function.

For more information about the `modifier` namespace interfaces, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

monitoring namespace

The `monitoring` namespace contains the `MessageObserver` [class](#) and the `model` namespace.

MessageObserver class

The `MessageObserver` class implements an observer. When creating an observer, you must specify the following parameters:

- [MessageEndpoint](#) of the message bus that provides a subscription for the message from the entities.
- Function for processing the received messages.
- [UID](#) of the observer.

You can use the methods of the `MessageObserver` class, you can subscribe an observer to a certain set of entities and a certain [message type](#). At the end of each neural [Network](#) execution cycle, the observer receives messages by [Subscription](#) and processes them in accordance with the specified function.

model namespace

The `model` namespace implements model monitoring functions:

[add_aggregated_spikes_logger\(\)](#)

It adds to the model an observer that collects information about spikes from specified senders at a specified frequency and writes the information it receives to the output stream in tabular format. Each row in the table is a sequence of numbers corresponding to the number of the model execution step at which the spike was generated and the number of spikes for each sender.

[Example of an add_aggregated_spikes_logger\(\) function record](#) ²

```
Index, Population1, Population2, Population3
0, 5, 3, 8
15, 2, 7, 1
17, 15, 4, 9
```

[add_projection_weights_logger\(\)](#)

Adds to the model an observer that collects, at a specified frequency, information about synapse weights in a specified projection and writes to the output stream, in a structured format, the number of the model execution step, the index of the neuron that generated the spike, the current values of the weights, and the last moments of their update, separated by the | (pipe) symbol.

[Example of an add_projection_weights_logger\(\) function record](#)

```
Step: 1
Neuron 1
0.123|50 0.456|45 0.789|40
```

[add_spikes_logger\(\)](#)

Adds an observer to the model that collects information about spikes in `SpikeMessages` from specified senders and, for each spike, writes to the output stream line-by-line the number of the model execution step at which the spikes were generated, the name of the spike sender, and the indices of the neurons that generated the spikes.

[Example of an add_spikes_logger\(\) function record](#)

```
Step: 1
Sender: Population1
15 23 42 57 89
Step: 2
Sender: Population1
5
```

[add_status_logger\(\)](#)

Adds to the model an observer that monitors `SpikeMessages` from all neuron populations in the model and writes to the output stream the number of received messages per model execution step.

For more information about the `monitoring` namespace, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

normalization namespace

The `normalization` namespace provides the functions and functors for normalizing the parameters of synapses and neurons in neural `Networks`. These functions and functors ensure stable performance of the `Model` and prevent problems related to unbalanced data.

Functions of the normalization namespace

`normalize_synapses()`

Normalizes the attributes of synapses in a specific `Projection` or in each projection of the relevant neural `Network`. This function iteratively processes each synapse in a projection. For each synapse, the function receives the value of the relevant attribute, adjusts its value according to the defined function, and sets the adjusted value for the attribute.

`normalize_neurons()`

Normalizes the attributes of neurons in a specific `Population` or in each population of the relevant neural `Network`. This function iteratively processes each neuron in a population. For each neuron, the function receives the value of the relevant attribute, adjusts its value according to the defined function, and sets the adjusted value for the attribute.

The functions of the `normalization` namespace are template functions, therefore you can use them when working with any type of neuron or synapse to normalize the values of relevant attributes.

Functors of the normalization namespace

`PotentialAccessor`

Reads and updates the `potential_` attribute while normalizing neurons. This attribute stores the value of the neuron membrane potential. The functor is a template functor, therefore you can use it for any type of neuron.

`WeightAccessor`

Reads and updates the `weight_` attribute while normalizing synapses. This attribute stores the value of the synapse weight. The functor is a template functor, therefore you can use it for any type of synapse.

`Rescaler`

Scales attribute values linearly from one range to another. The functor is a template functor and supports any numeric data type.

If necessary, you can implement and use your own functors to access the attributes of neurons and/or synapses, and to normalize their values.

For more information about the functions and functors of the `normalization` namespace, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

output namespace

The `output` namespace implements an interface for accessing the output channel and `SpikeMessage` converters. `SpikeMessages` contain spikes. 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 accepts spike messages from the associated `Population`. Spike messages are transmitted to the output channel over several clock cycles of the neural network. When creating an output channel, you must specify the unique `UID` of the output channel and the `MessageEndpoint`.

The messages of the `SpikeMessage` type received by the output channel can be converted to one of the following data formats:

- 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 indices of neurons that generated spikes from a sequence of the `SpikeMessage` messages.

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

SpikeMessage converters

The `output` namespace provides the following objects to convert received `SpikeMessages`:

`ConvertToSet` class

Implements a data converter that receives messages of the `SpikeMessage` type and converts them to a vector of indices of neurons that generate spikes.

`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

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.

For more information about the functions and interfaces of the `output` namespace, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

population namespace

The `population` namespace contains the `creators` and `neurons_generators` namespaces. Namespaces provide the functions and interfaces needed to generate populations and populate them with a specific type of neurons.

If necessary, you can use the `Population` class constructor to create populations and neurons.

creators namespace

The `creators` namespace implements functions for generating neuron populations:

`from_container()`

Generates populations from a container containing attributes of neurons.

`make_random()`

Generates populations with random values of neuron attributes.

`make_default()`

Generates populations with the default neuron attribute values.

`make_clone()`

Generates populations whose neuron attribute values are copied from the attributes of a specific neuron.

neurons_generators namespace

The `neurons_generators` namespace implements functions and interfaces for generating neurons:

`from_container ()` function

Generates neurons from a container containing attributes of neurons.

`MakeRandom ()` constructor

Generates neurons containing random attribute values.

`make_default ()` function

Generates neurons containing the default attribute values.

`make_clone ()` function

Generates neurons whose attribute values are copied from the attributes of a specific neuron.

For more information about the functions and interfaces of the `population` namespace, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

projection namespace

The `projection` namespace contains the `creators`, `synapse_generators`, and `parameters_generators` namespaces. Namespaces provide the functions and interfaces needed to generate projections and populate them with a specific type of synapses. Before calling the functions, you must first create the populations that need to be linked by the projection.

The namespace also provides functions that let you create a new projection by linking two populations in one of the following ways:

- Randomly
- Completely (each neuron in the presynaptic population is linked to each neuron in the postsynaptic population)

The `SpikeMessage` handler from the `modifier` namespace that implements the [Winner Takes ALL \(WTA\) mechanism](#) can also be added to the `ModelExecutor` by using the namespace functions.

If necessary, you can use the methods provided by the `Projection` class to create projections and synapses.

creators namespace

The `creators` namespace implements functions for generating synapse projections. These functions let you generate synapse projections by connecting neurons as follows:

- Complete connection: each neuron in the presynaptic population is connected to each neuron in the postsynaptic population.
- Serial connection: neurons in a smaller population are connected to neurons in another population in a specific order.
- Exclusive connection: neurons of populations do not connect if their indices match.
- One-to-one: neurons in the presynaptic and postsynaptic populations are connected in a one-to-one relationship.
- Random connection: neurons of populations connect according to a certain probability.
- Constrained random connection: each neuron from one population connects to a certain number of random neurons of another population.
- Conditional connection: neurons are connected if the synapse generation function returns a non-zero `std::optional` value.

You can also generate synapse projections in the following ways:

- From a container containing synapse attributes and presynaptic and postsynaptic neuron indices.
- From the `std::map` object, in which the keys are the indices of the presynaptic and postsynaptic neurons, and the values are the synapse attributes.
- Copy synapses from another projection.

synapse_generators namespace

The `synapse_generators` namespace implements functions and interfaces that you can use to generate synapses by connecting neurons in the following ways:

- Complete connection: each neuron in the presynaptic population is connected to each neuron in the postsynaptic population.
- Serial connection: neurons in a smaller population are sequentially connected to neurons in another population.
- Exclusive connection: each neuron of one population connects to each neuron of another population, except for those whose indices match.
- One-to-one: neurons of populations are connected in a one-to-one relationship.
- From a container containing synapse attributes and presynaptic and postsynaptic neuron indices.
- From an `std::map` or `std::unordered_map` object containing synapse attributes as the value and presynaptic and postsynaptic neuron indices as the key.

- Random connection: each neuron in the presynaptic population is connected to each neuron in the postsynaptic population according to a certain probability.
- Constrained random connection: each neuron from one population connects to a certain number of random neurons of another population.
- Conditional connection: specific neurons of populations are connected based on the synapse generation function.
- Copy the structure of another projection.

parameters_generators namespaces

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

For more information about the `projection` namespace functions, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

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()`

Loads the neural network. The neural network topology must be described in the `SONATA` format.

`save_network()`

Saves the neural network topology and weights in `SONATA` format.

For more information about the `sonata` namespace functions, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

storage namespace

The `storage` namespace contains the `native` namespace, which implements functions for loading and saving `SpikeMessage` messages in the `basic KNP format`. The namespace provides the following functions for working with basic format files:

`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.

For more information about the `storage` namespace functions, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

synchronization namespace

The `synchronization` namespace implements the `get_network_copy(.)` function for getting the current state of the neural `Network` from the `Backend`. For example, you can use this function whenever the quality of neural network predictions changes or after every few execution steps. You can compare the quality of the saved neural network copies to select the best one for downloading to the device.

BackendLoader class

The `BackendLoader` class implements the `Backend` instance loader. The pointer to the backend instance is passed to the `ModelExecutor` to execute the model in the backend.

Using the methods of the `BackendLoader` class, you can download dynamic libraries and check whether the downloaded libraries are backends.

For more information about methods of the `BackendLoader` class, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

The `BackendLoader` class is implemented using the [Boost:DLL](#)  library.

Model class

The `Model` class implements a model. A model stores a neural `Network` and an `InputChannel` and `OutputChannel`, and links them to each other.

When creating a model, you must specify a previously created neural `Network`. After you create a model, you must add an `InputChannel` and `OutputChannel` to it. You can add input and output channels by connecting them to the `Projections` and `Populations` used in the neural network.

You can also use the `Model` class methods to get the neural network and input and output channels used in the model, and to get the base data of the model.

To execute a model on different backends, you can load the model into a `ModelExecutor` and call the `start()` method.

For more information about methods of the `Model`, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

ModelExecutor class

The `ModelExecutor` class implements a model executor. When creating a model executor, you must pass the `Model`, the `Backend` instance that will execute the model, and the `InputChannel` map. Each model executor contains the following attributes:

- `ModelLoader`
- `MessageObservers`

Using the `ModelExecutor` class methods, 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. In addition to executing the model, you can use methods of the `ModelExecutor` class to do the following:

- Add a previously created `MessageObserver` to the `ModelExecutor`.
A `MessageObserver` must be added to receive messages by [Subscription](#).
- Add the `SpikeMessage` handler from the `modifier` namespace to the `ModelExecutor`.
- Start and stop `Model` training.
- Get the `ModelLoader` and the pointer to the `Backend` instance.

For more information about methods of the `ModelExecutor` class, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

ModelLoader class

The `ModelLoader` class implements a model loader that loads the `Model` into the relevant `Backend` instance. When creating a model loader, you must specify an `InputChannel` map and a pointer to the `Backend` instance. After the model loader is created, you can call the `load()` method to load the model into the backend. The model must first be created while passing the neural `Network` storing the neuron `Populations` and synapse `Projections` to the model constructor.

The `ModelLoader` class provides the methods that you can use to get the `InputChannel` and `OutputChannel` that are used in the `Model` and the pointer to the loaded `Backend` instance.

Each model loader contains the following attributes:

- `Backend` instance where the model is loaded.
- `InputChannel` map containing the `UID`, `hashed ID`, and `input data converter`.
- `InputChannel` and `OutputChannel`.

For more information about methods of the `ModelLoader`, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

Network class

The `Network` class implements a neural network. `Populations` and `Projections` are stored in a neural network. All populations of a network are connected by projections.

The `Network` class provides methods that let you do the following:

- Add previously created `Populations` and `Projections` to a neural network.
- Remove unnecessary `Populations` and `Projections` with defined `UIDs` from a neural network.
- Interlink `Populations` and `Projections` that have been added to a neural network.
- Check whether a neural network contains a `Population` with a specific unique identifier.
- Check whether a neural network contains a `Projection` with a specific unique identifier.
- Get `Populations` and `Projections` that are part of a neural network.
- Get the `BaseData` of a neural network.
- Get information about the number of `Populations` and `Projections` within a neural network.

After you have added `Populations` and `Projections` to the neural network and linked them together, you can pass the neural network to a `Model` to create it.

For more information about methods of the neural `Network`, please refer to the [Kaspersky Neuromorphic Platform API documentation](#).

Solving common tasks in C++

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

Additionally, you can find the following examples of using the platform in `/usr/src/knp/examples` directory after [installing](#) `knp-examples` package or in the platform repository.

- `mnist-client` is an example of classifying images of handwritten numbers from the MNIST database.
- `mnist-learn` is an example of training a neural network on images of handwritten numbers and their labels from the MNIST database and its execution on a multi-thread CPU backend. The neural network example uses the CoLaNET architecture.
- `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.

The directory with each example contains a `README.md` file with a description of the example in it.

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 necessary, you can add a new type of neuron to the [neuron_traits](#) library.

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

To add a new neuron type:

1. Go from the working directory with the platform source code 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 necessary, you can add a new type of synapse to the [synapse traits](#) library.

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

To add a new synapse type:

1. Go from the working directory with the platform source code 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.

Preparing the model and executing it on the backend

To execute a model on the backend, you must first create the necessary platform objects and link them together.

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

The model preparation scenario consists of the following steps:

1 Creating populations

To create populations, you can use the `Population` [class](#) or the `population` [namespace functions](#). When creating a population, a neuron generator is passed to the population constructor or population generation function. If required, you can remove generated neurons from the populations or add new neurons to them.

2 Creating projections

To create projections, you can use the `Projection` [class](#) or the `projection` [namespace functions](#). When created, a synapse generator is passed to the projection constructor or projection generation function. If necessary, you can remove generated synapses from projections or add new synapses to them.

3 Creating a neural network

Create an instance of the `Network` [class](#).

Use the `add_population()` and `add_projection()` methods of the `Network` class to add the previously created populations and projections to the neural network. If necessary, you can use the `connect_populations()` method to additionally interconnect the populations added to the neural network.

4 Creating a model

Create an instance of the `Model` [class](#). When creating a model, pass the previously created neural network to the constructor.

The scenario for executing a model on the backend consists of the following steps:

1 Loading the backend

Create a backend loader by using the `BackendLoader` class. To load the backend, use the `load()` method while passing to it the path to the backend dynamic library. The method returns the pointer to the `Backend` instance that needs to be passed to the `ModelExecutor`.

After the backend is loaded, it is automatically initialized. When the backend is initialized, `MessageBus` objects and `MessageEndpoints` for connecting to the message bus are automatically created.

If necessary, you can execute multiple different models on a loaded backend. You can also load multiple backends to run one or more models on them.

2 Starting model execution

To execute a model, you must first create a model executor using the `ModelExecutor` class. When creating a model executor, pass to it the previously created `Model`, the pointer to the loaded `Backend` instance, and the `InputChannel`. If necessary, create a `MessageObserver` and add it to the `ModelExecutor` to receive messages by subscription.

After a `Model` is loaded into the `ModelExecutor`, the following steps are performed automatically:

1. The `Model` is loaded into the `Backend`.
2. The framework creates `InputChannel` and `OutputChannel` objects and adds them to the `Model`.
3. The `Backend` automatically subscribes model objects to each other.

To start execution of the model, use the `start()` method. When executing a model on the backend, messages are automatically transmitted to the message endpoint via the message bus.

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

#include <vector>

using Backend = knpp::backends::single_threaded_cpu::SingleThreadedCPUBackend;
using DeltaProjection = knpp::core::Projection<knpp::synapse_traits::DeltaSynapse>;
using BLIFATPopulation = knpp::core::Population<knpp::neuron_traits::BLIFATNeuron>;
using Population = knpp::backends::single_threaded_cpu::SingleThreadedCPUBackend::PopulationVariants;
using Projection = knpp::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 knpp::neuron_traits::neuron_parameters<knpp::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, knpp::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, knpp::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 [UID](#) of the associated [Population](#) and 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 [UID](#) 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 [MessageEndpoint](#) object for connecting to the message bus.

Example of creating a message bus endpoint:

```
C++

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

10. Connect the [InputChannel](#) to the input [Projection](#), and connect the [Population](#) to the [OutputChannel](#). 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 in the example 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 in the example.

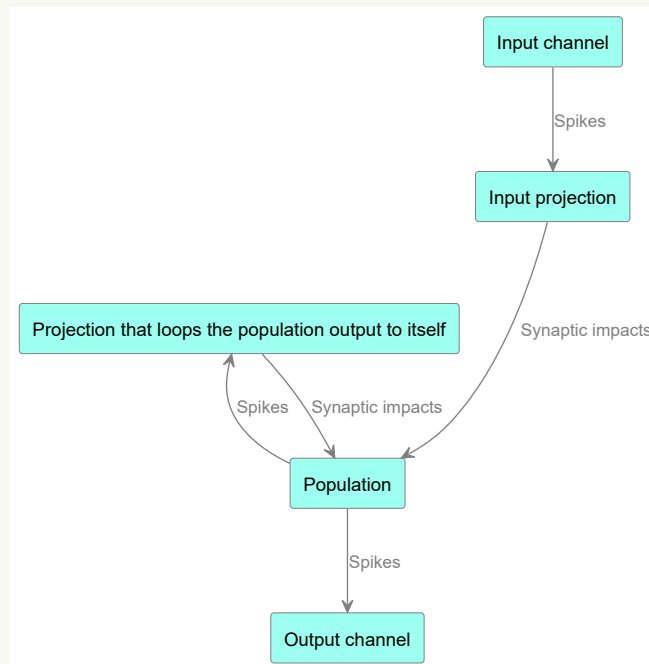


Diagram of the neural network execution in the example

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 `AltAILIF` model
- Neuron based on the `BLIFATNeuron` model
- Neuron based on the `SynapticResourceSTDPBLIFATNeuron` model

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
- Synapse based on the `SynapticResourceSTDPDeltaSynapse` model

If needed, you can implement and use your own synapse models when [working with source code](#).

core module

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
<code>messaging</code>	Module containing interfaces to messages.
<code>Backend</code>	Base class for specific backend implementations.
<code>BaseData</code>	Structure defining a set of base data.

Component	Description
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	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.

The figure below shows how the module components interact.

In the Kaspersky Neuromorphic Platform, the `BaseData` structure contains basic information about an object and can contain `UID` and `TagMap`.

Objects of the `Population` and `Projection` classes store data that the backend processes and passes to other platform objects via messages implemented in classes of the `messaging` module. Messages contain the sender's `UID` in the header. Message exchange is implemented using the methods of the `Backend` class, which stores the `MessageBus` and `MessageEndpoint` classes, as well as a `Device`. To connect to the `MessageBus`, platform objects create `MessageEndpoints` that are used by the message bus to interact with these objects during the message exchange process. Each `MessageEndpoint` has a container of the `Subscription` class.

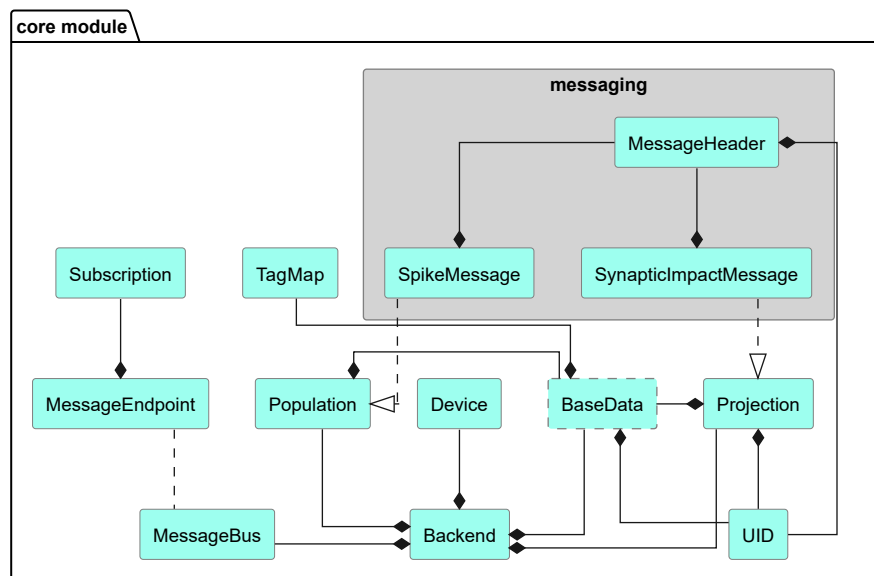


Diagram of interactions between components in the core module

messaging module

The `messaging` module implements interfaces for accessing messages. Each message has the following property set:

- `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.

Message exchange is provided by the `MessageBus` message bus. 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
<code>SpikeMessage</code>	A message to a synapse projection containing a spike . More than one projection can be subscribed to the messages containing spikes.	<code>header</code> —message header containing the UID of the input channel or population whose neurons generated the spikes and the time when the spikes were generated. <code>neuron_indexes</code> —unique indices of the neurons in the population that generated spikes.
<code>SynapticImpactMessage</code>	A message from a synapse projection to the neuron population containing synaptic impact .	<code>header</code> —message header containing the projection UID and the time when the message was sent. <code>postsynaptic_population_uid</code> — UID of the postsynaptic population. <code>Presynaptic_population_uid</code> — UID of the presynaptic population. <code>is_forcing</code> —flag to enable or disable plasticity in a synapse projection. If the synapse projection uses synaptic plasticity, the attribute is <code>true</code> , otherwise it is <code>false</code> . <code>impacts</code> —array containing the following attributes: • <code>connection_index</code> — connection index. • <code>impact_value</code> — synaptic impact value. • <code>synapse_type</code> — type of synapse. • <code>presynaptic_neuron_index</code>—unique index of the presynaptic neuron from which the spike was received by the projection synapse. • <code>postsynaptic_neuron_index</code>—unique postsynaptic neuron index. The value of the synaptic impact is sent to this neuron to change the values of its attributes. The <code>impacts</code> array may contain more than one set of the listed attributes to convey synaptic impacts to multiple neurons in the same population.

Kaspersky Neuromorphic Platform provides methods for interacting with messages. For example, you can use the `send message` method of the `MessageEndpoint` class to send a specific message through a message endpoint to a message bus.

Backend class

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

- Manage 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.

If you have previously created a [model](#) that has been passed a [neural network](#) that stores [populations](#) and [projections](#), you can use the [ModelLoader](#) class's load model method to load it into the backend before its initialization. You can also manually load projections containing [synapses](#) and populations containing [neurons](#). To do so, you can use the load projections and populations methods of the `Backend` base class or the corresponding methods of a specific backend class.

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](#). If necessary, you can subscribe the required platform objects to [messages](#) by using the subscribe method.

The `Backend` base class provides methods that you can use to manage the training and execution of the neural network.

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.

BaseData structure

The `BaseData` structure defines a set of base data that is common to some platform 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

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.

Device class

The `Device` class is the base class for the device classes. The class inheritors override methods 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.

MessageBus class

The `MessageBus` class implements an interface for accessing a message bus. The message bus provides for the receipt, distribution and delivery of messages between objects connected to it.

The `MessageBus` class provides methods that let you do the following:

- Create message bus endpoints.
- Receive, distribute and deliver messages between objects connected to the message bus.

A message bus provides a set of [message endpoints](#) through which platform objects and [external environment](#) objects can subscribe to [messages](#) or send messages. You can create message bus endpoints and then subscribe them to certain types of messages by using the subscribe method of an instance of the `MessageEndpoint` class. 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](#). You can create a message endpoint by using the corresponding method of the `MessageBus` class. A message bus can have multiple message endpoints.

The message endpoint stores the message recipient ID and the subscription class key that is used to filter [messages](#). A message endpoint can store a set of subscriptions to messages from different senders and for different recipients.

The `MessageEndpoint` class provides methods that let you do the following:

- Subscribe message endpoints to messages of a specific type.
- Unsubscribe from messages from specific senders.

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

- Receive messages from a sender to which a message endpoint is subscribed.

Before calling the method, you must use the message routing method of the `MessageBus` class object for the message endpoint to receive messages.

- Send messages via a message bus.

Set of Population classes

The set of `Population` classes implements containers, each of which contains [neurons ?](#) of the same model. To initialize a population, you must specify the number of neurons and the generator of the specific type of neurons.

You can upload populations to the backend to compute the neural network. If a model storing populations is loaded on the backend, populations are automatically loaded using the [ModelLoader](#). You can also manually pass population data to the backend using the load populations method of the [Backend](#) base class.

After computation, the new population properties are stored in the backend. To get the current neural network parameters from the backend, you can call 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.

The `Population` class set provides methods that let you do the following:

- Add neurons to a previously created population.
- Remove one or more neurons from a population.
- Get the unique ID of a population.
- Get the attributes of population neurons.
- Get information about the number of neurons in a population.
- Perform iteration for a population.
- Get a certain element of a population.

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 identifiers of these populations must be specified when initializing the 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.

The `Projection` class set provides methods that you can use to get the unique identifier of a projection and the attributes of the projection synapses. You can also get information about the number of synapses in a projection, perform iteration for a projection, and get a specific element of a projection.

Set of Subscription classes

The `Subscription` class set implements the functionality for subscribing to messages and stores the received [messages](#). The Python framework provides the following subscription classes for storing messages of a certain type until they are requested by the receiver:

- `SpikeMessageSubscription`: for subscribing to `SpikeMessages`.
- `SynapticImpactMessageSubscription`: for subscribing to `SynapticImpactMessages`.

You can create a subscription by using the `subscribe` method of the `MessageEndpoint` class. Created subscriptions are stored in message endpoints. If you delete a message endpoint that is associated with subscription objects, these subscriptions will also be deleted. Messages under those subscriptions stop being transmitted.

A subscription provides for [filtering of messages](#). The subscription object defines the identifiers of message recipients and senders that are used by the message bus to distribute messages and deliver them to the recipients.

The `Subscription` class set provides methods that let you do the following:

- Add senders to a subscription.
- Remove senders from a subscription.
- Check whether a subscription is associated with a specific sender.

TagMap class

An object of the `TagMap` class 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. An object of the `TagMap` class can be used as part of the [BaseData](#) structure.

UID class

The `UID` class implements a unique identifier for an object. The object identifier is often used as part of the [BaseData](#) structure.

If necessary, you can generate a random UID or create a unique [UUID](#), and get the hash value of an object identifier. The `UID` class also provides logical, comparison, and assignment operators.

uid_hash class

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

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.

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-2 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-2 neuromorphic processor. Resulting neural networks are adapted for execution on the AltAI-2 neuromorphic processing unit, or a software emulator of the latter.

Python framework is a set of units that manage backends, transform the neural network representation into various formats, and select the optimal neural network to run.

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

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.
<code>knp</code>	A package that implements a Python framework for an AltAI SNN backend.
<code>input</code>	A module that implements an interface for accessing the input channel.

Component	Description
output	A module that implements an interface for accessing the output channel.
sonata	A module 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 <code>SynapticImpactMessage</code> 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 <code>SpikeMessage</code> observer.

The figure below shows how the Python framework components interact.

In Kaspersky Neuromorphic Platform, a `Network` object can be loaded via the `sonata` namespace function. Neural network, as well as the input and output channel containers implemented in the `input` and `output` modules, respectively, are stored in an object of the `Model` class. Input and output channels are used to receive messages containing spikes from the input data stream and neural network populations.

You can execute a model using the `start()` method of the `ModelExecutor` class object. A model executor also contains `SpikeMessageObservers` and `ImpactMessageObservers` that process received messages according to the defined function, and it contains a `ModelLoader` class object whose methods can be used to load a model in a specific backend instance. To execute a model in a backend instance, the backend object received from the `BackendLoader` class object is passed to the model executor.

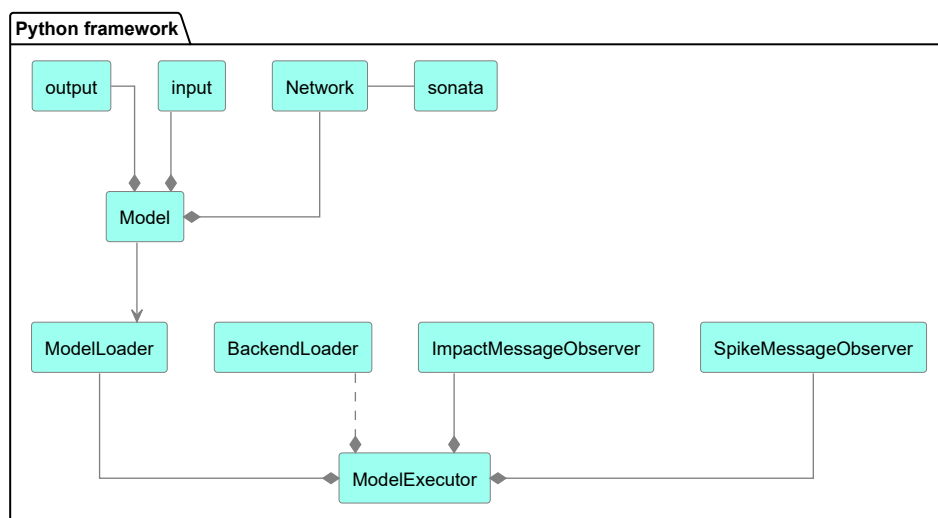


Diagram of interaction of Python framework components

ANN2SNN package

The `ANN2SNN` package is intended for creating neural networks that can be converted into [spiking neural networks](#) [\[2\]](#) and placed for execution on the AltAI-2 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-2 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 $-16,384$ to $16,383$.

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-2 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-2 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-2 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.
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-2 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).

input module

The `input` module implements an interface for accessing the `InputChannel`, which accepts [SpikeMessages](#). When initializing an input channel, you must specify a unique [UID](#) for the input channel, a [message endpoint](#) to the message bus that is used to send data to subscribers, and a functor for generating `SpikeMessages`. If a [subscription](#) is present, generated messages can be transmitted to subscribers via message endpoints.

If you have loaded a [model](#) storing a [neural network](#) and input and output channels into a [model executor](#), the sent messages are automatically transmitted when the model is executed on the backend.

output module

The `output` module implements an interface for accessing `OutputChannel`. The output channel receives a set of messages containing spikes from an associated population over several neural network execution steps.

When initializing an output channel, you must specify the unique [UID](#) of the output channel and the [message endpoint](#) for connecting to the message bus.

Messages received by the output channel must be converted to the data format that you need.

sonata module

The `sonata` module implements functions for loading and saving a neural network in the [SONATA](#) format. The namespace provides the following functions:

`load_network()`

Loads the neural network.

`save_network()`

Saves the neural network topology and weights.

BackendLoader class

The `BackendLoader` class implements a backend instance loader. The backend instance is passed to the [ModelExecutor](#).

The `BackendLoader` class provides methods that let you do the following:

- Load backends.
- Check whether the loaded libraries are backends.

ImpactMessageObserver class

The `ImpactMessageObserver` class implements an observer that subscribes to [SynapticImpactMessages](#). When initializing an observer, you must specify a message endpoint for connecting to the message bus that provides a subscription to messages, a function to handle received messages, and the observer identifier.

You can use the `subscribe` method to subscribe an observer to `SynapticImpactMessages`. At the end of each neural network execution cycle, the observer receives messages by [subscription](#) and processes the messages in accordance with the specified function.

Model class

The `Model` class implements a model. The model stores connections between [input](#) and [output](#) channels and [populations](#) and [projections](#). When initializing the model, you must specify a previously created neural network. After a model is created, you must add input and output channels to it.

The `Model` class provides methods that let you do the following:

- Add input and output channels to a model.
- Get the neural network used in a model.
- Get the input and output channels used in a model.
- Get the unique ID of a model.

To execute a model on different backends, you can download the model into an object of the [ModelExecutor](#) class and call the model execution method.

ModelExecutor class

The `ModelExecutor` class implements a model executor. When initializing a model executor, you must specify the [model](#) to execute, the [backend](#) instance that will execute the model, and the input channel map.

The `ModelExecutor` class provides methods that let you execute a loaded model. To execute a model, the `ModelExecutor` class uses a backend instance obtained from a [BackendLoader](#) class object. In addition to executing the model, you can use methods of the `ModelExecutor` class to also do the following:

- Add a message observer for a specific type of messages to the model executor.
- Start and stop training a model.
- Get an instance of the backend used to execute the model.
- Get the output channel.

ModelLoader class

The `ModelLoader` class implements a model loader that uploads the [model](#) to the appropriate [backend](#) instance.

The `ModelLoader` class provides methods that let you do the following:

- Create model loaders.
- Load a model on the backend.
The model must first be created by passing a [neural network](#) storing neuron [populations](#) and synapse [projections](#) to the model constructor.
- Get the [input](#) and [output](#) channels used in a model.

Network class

The `Network` class implements a neural network. A neural network stores [populations](#), [projections](#), and a shared interface for accessing individual neurons and synapses within the network. In a neural network, all populations are connected by projections.

Each neural network has the following properties:

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

The `Network` class provides methods that let you do the following:

- Add populations and projections to a neural network.
- Delete populations and projections from a neural network.
- Get a unique neural network ID.
- Get populations and projections as part of a neural network.
- Get ranges of population iterators and projections in a neural network.
- Get information about the number of populations and projections in a neural network.

SpikeMessageObserver class

The `SpikeMessageObserver` class implements an observer that subscribes to [SpikeMessages](#). When initializing an observer, you must specify a message endpoint for connecting to the message bus that provides a subscription to messages, a function to handle received messages, and the observer identifier.

You can use the subscribe method to subscribe an observer to `SpikeMessages`. During the neural network execution cycle, the observer receives messages by [subscription](#) and processes the messages in accordance with the specified function.

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 step number 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 step number when the spike arrived at the output channel.
        print(f'STEP {step}')
```

Neural network execution steps [?](#)

The neural network in the example 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 in the example.

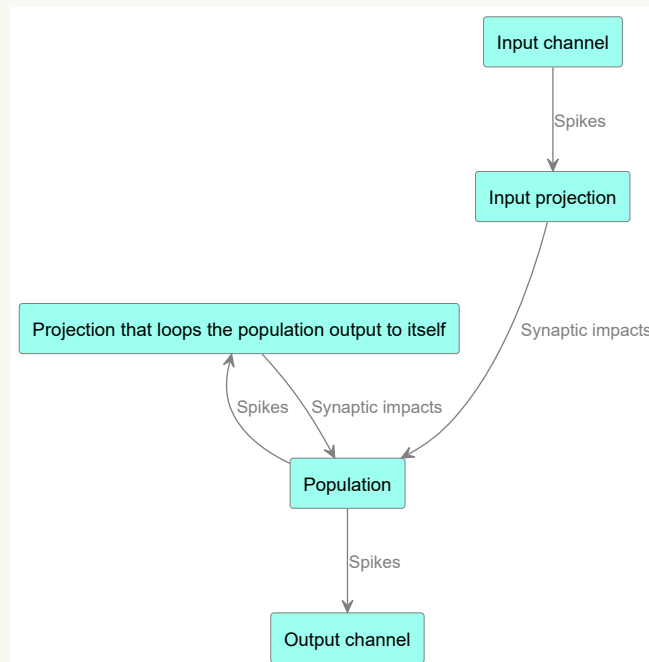


Diagram of the neural network execution in the example

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-2 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-2 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.keras -l mnist_placer.log -o mnist.json
```

3 Starting inference of the neural network

Connect the backplane with the AltAI-2 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-2 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_checkpoint_callback = ModelCheckpoint(
    filepath="mnist.keras",
    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_checkpoint_callback])
# Load the trained model.
model = load_model(
    "mnist.keras",
    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-2 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-2 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-2 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-2 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-2 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-2 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-2 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-2 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-2 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 populations of neurons and connections between them.
- Create and train neural networks.
- Manage neural network hyperparameters.
- Created new neuron models.
- Create new topologies of neural networks.
- Implement application solutions.



[OPEN the C++ API REFERENCE \(English\)](#)

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, load the current version of the remote platform repository into 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

Kaspersky Neuromorphic Platform Terms of Use

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 a CPU single-threaded backend dynamic library.
- `knp-cpu-multi-threaded-backend_<knp_version>_amd64.deb`: a deb package with a CPU multi-threaded backend dynamic library.
- `knp-gpu-cuda-backend_<knp_version>_amd64.deb`: a deb-package with a backend library for GPUs with support for CUDA technology.
- `knp-cpp-framework_<knp_version>_amd64.deb`: a deb package with a C++ framework dynamic library.
- `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-gpu-cuda-backend-dev_<knp_version>_amd64.deb`: a dev-deb package for `knp-gpu-cuda-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 containing the API reference.
- `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.

Terms of Use of the backends for the AltAI-2 neuromorphic processor and the ANN2SNN package

The [AltAI-2](#) 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 an `AltAI SNN` backend dynamic library for the AltAI-2 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 an `AltAI ANN2SNN` backend dynamic library.
- `knp_ann2snn_x86_64-<motiv_version>-py3-none-any.whl`: a whl package containing an `ANN2SNN` package and an `AltAI ANN2SNN` backend dynamic library 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 dynamic library to install on an ARM-based computer.

where `<motiv_version>` is the version of the backends for the AltAI-2 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-2 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-2 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.

Troubleshooting

This section contains a description of possible malfunctions in the operation of Kaspersky Neuromorphic Platform and how to fix them.

The platform is not being built with the Python framework

Issue

The platform is not being [built with the Python framework](#).

Solution

If stubgen is not available on the device, the `KNP_PYTHON_FRAMEWORK_BUILD` configuration setting is automatically set to `OFF`, which causes the platform to be built without the Python framework. When configuring the build, set `KNP_PYTHON_FRAMEWORK_BUILD` to `ON`.

The compiler or build system cannot find the Boost library in Windows

Issue

The compiler or build system cannot find the Boost library in Windows. The path to the library is [specified in the `Boost\_ROOT` setting in the `CMakePresets.json` file](#).

Solution

The compiler or build system can use the old value of the `Boost_ROOT` setting that is stored in the cache. In this case, you must delete the old value of the setting from the cache. To do so, in Visual Studio, on the **Project** tab, do one of the following:

- Select **Delete cache and reconfigure**. The old setting value will be removed from the cache.
- Select **View CMakeCache.txt** and clear the `Boost_ROOT` setting.

After removing the old setting value from the cache, rerun the [build configuration](#).

Applications

This section provides additional information that supplements the primary text of the document.

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.

Structure and example of a basic format file

A basic format file describing input or output data contains the following parameters:

- `attributes`: array of basic format file attributes containing a set of "key-value" pairs with the following keys:
 - `name`: name of the file attribute. The following attribute names are provided: `magic` and `version`. The `magic` attribute corresponds to a number that identifies the file format. The `version` attribute corresponds to the format version.
 - `shape`: size of the array specified in the `value` parameter for the `version` attribute.
 - `type`: set of "key-value" pairs describing the value of a file attribute using the following keys:
 - `class`: type of file attribute value.
 - `size`: possible range of the file attribute value.
 - `endianness`: order in which the bytes of a file attribute value are stored.
 - `value`: value of the file attribute.
- `spikes`: parameter that describes the generated spikes using the following parameters:
 - `attributes`: array of attributes describing how the generated spikes are sorted; contains a set of "key-value" pairs with the following keys:
 - `name`: name of the sorting attribute. The parameter has the `sorting` value.
 - `type`: set of "key-value" pairs describing possible sorting methods using the following keys:
 - `class`: data type that describes possible sorting methods. The parameter has the `Enumeration` value.
 - `mapping`: list of possible ways to sort spikes and their integer values. The following sorting methods are provided: `by_id`, `by_time` and `none`. The `by_id` method corresponds to sorting by the indices of neurons that generated spikes. The `by_time` method corresponds to sorting by spike generation timestamps. The `none` method corresponds to no sorting.
 - `value`: selected method of sorting spikes.
 - `node_ids`: indices of the neurons that generated spikes. The parameter contains a set of "key-value" pairs with the following keys:
 - `shape`: size of the array specified in the `value` parameter.
 - `type`: set of "key-value" pairs that describe the neurons that generated the spikes using the following keys:
 - `class`: type of neuron index value.

- `size`: possible range of neuron index values.
- `endianness`: order in which neuron index bytes are stored.
- `value`: array of UID indices of neurons that generated spikes.
- `timestamps`: timestamps of spike generation. The parameter contains a set of "key-value" pairs with the following keys:
 - `attributes`: array of parameters describing the spike generation timestamps. The array contains the `values` parameter, which corresponds to the timestamp value type. The `values` parameter has the `step` value, which corresponds to the neural network execution steps. In addition to the `values` element, the `attributes` array contains a set of "key-value" pairs with the following keys:
 - `name`: name of the timestamp attribute. The parameter has the `units` value.
 - `type`: set of "key-value" pairs describing timestamps. The `class` key corresponds to the type of value specified for the `values` parameter. The `charSet` key corresponds to the standard used to encode timestamp values.
 - `shape`: size of the array specified in the `value` parameter.
 - `type`: set of "key-value" pairs describing timestamp values using the following keys:
 - `class`: type of timestamp value.
 - `endianness`: order in which the bytes of timestamp values is stored.
 - `value`: spike generation timestamps.

An example of a basic format file in JSON notation is provided below.

```
{
  "attributes": [
    {
      "name": "magic",
      "type": {
        "class": "Integer (unsigned)",
        "size": 32,
        "endianness": "little-endian"
      },
      "value": 2682
    },
    {
      "name": "version",
      "shape": [2],
      "type": {
        "class": "Integer (unsigned)",
        "size": 32,
        "endianness": "little-endian"
      },
      "value": [0, 1]
    }
  ],
  "spikes": {
    "attributes": [
      {
        "name": "sorting",
        "type": {
          "class": "Enumeration",
          "mapping": {
            "by_id": 1,
            "by_time": 2,
            "none": 0
          }
        },
        "value": "by_time"
      }
    ]
  }
}
```

```

    }
  },
  "node_ids": {
    "shape": [5],
    "type": {
      "class": "Integer (unsigned)",
      "size": 64,
      "endianness": "little-endian"
    },
    "value": [1, 2, 3, 4, 5]
  },
  "timestamps": {
    "attributes": [
      {
        "name": "units",
        "type": {
          "class": "String",
          "charSet": "ASCII"
        }
      }
    ],
    "values": "step"
  },
  "shape": [5],
  "type": {
    "class": "Float",
    "endianness": "little-endian"
  },
  "value": [1, 2, 3, 4, 5]
}
}
}

```

Model quality evaluation results transmitted in CSV format

When calling the method for writing the model evaluation results for the [InferenceResultsProcessor](#) class, the following data is written to the data stream in CSV format:

- **CLASS** is the index of the object class.
- **TOTAL_VOTES** is the total number of votes for the class. This value corresponds to the sum of truly positive, falsely positive, and falsely negative results.
- **TRUE_POSITIVES** is the number of truly positive results for the class.
- **FALSE_NEGATIVES** is the number of falsely negative results for the class.
- **FALSE_POSITIVES** is the number of falsely positive results for the class.
- **TRUE_NEGATIVES** is the number of truly negative results for the class.
- **PRECISION** is the model precision relative to the results for the specific class of objects.
- **RECALL** is the model recall relative to the results for the specific class of objects.
- **PREVALENCE** is the model prevalence relative to the results for the specific class of objects.
- **ACCURACY** is the model accuracy relative to the results for the specific class of objects.
- **F_MEASURE** is the F score relative to the results for the specific class of objects.

Formulas for calculating model quality evaluation metrics

You can use the functions of `the_inference_evaluation` namespace to obtain the following model quality evaluation metrics:

- Precision
- Recall
- Prevalence
- Accuracy
- F-score

Precision

The metric is calculated using the formula shown in the figure below, where:

- `Precision` is the precision metric.
- `TP` is the number of true positives.
- `FP` is the number of false positives.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Formula for calculating precision

Recall

The metric is calculated using the formula shown in the figure below, where:

- `Recall` is the recall metric.
- `TP` is the number of true positives.
- `FN` is the number of false negative results.

$$\text{Recall} = \frac{TP}{TP + FN}$$

Formula for calculating recall

Prevalence

The metric is calculated using the formula shown in the figure below, where:

- `Prevalence` is the prevalence metric.
- `TP` is the number of true positives.

- FN is the number of false negative results.
- total is the total number of results.

$$\text{Prevalence} = \frac{\text{TP} + \text{FN}}{\text{total}}$$

Formula for calculating prevalence

Accuracy

The metric is calculated using the formula shown in the figure below, where:

- Accuracy is the accuracy metric.
- TP is the number of true positives.
- TN is the number of true negative results.
- total is the total number of results.

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{total}}$$

Formula for calculating accuracy

F-score

The metric is calculated using the formula shown in the figure below, where:

- F-score is the F-score metric.
- Precision is the precision metric.
- Recall is the recall metric.

$$\text{F-score} = \frac{2 * \text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

Formula for calculating the F-score

Settings and an example YAML file containing the configuration of the AltAI-2 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-2 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-2 neuromorphic processing unit backplane.
- `col` includes a `chips` setting. This `chips` value sets the number of modules in the AltAI-2 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-2 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

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

Build configuration parameters specific to the Kaspersky Neuromorphic Platform

In addition to the standard CMake configuration settings, you can use the settings specific to the Kaspersky Neuromorphic Platform when [configuring the build](#). These settings are described in the table below.

Configuration setting	Possible values	Description
<code>KNP_BUILD_CUDA</code>	The following values are accepted: <code>ON</code>: enables building a deb package with a backend library for GPUs with CUDA support. If the CUDA Toolkit software is available on the device, the setting is automatically set to <code>ON</code>. <code>OFF</code>: disables building a deb package with a backend library for GPUs with CUDA support. If the CUDA Toolkit software is not available on the device, the setting is automatically set to <code>OFF</code>.	This parameter regulates the build of a deb package with a backend library for GPUs with support for CUDA technology.
<code>KNP_BUILD_DOCUMENTATION</code>	The following values are accepted: <code>ON</code>: enables building a deb package with an API reference. <code>OFF</code>: disables building a deb package with an API reference.	This setting controls building a deb package with an API reference.
<code>KNP_BUILD_EXAMPLES</code>	The following values are accepted: <code>ON</code>: enables building a deb package with Kaspersky Neuromorphic Platform usage examples. <code>OFF</code>: disables building a deb package with Kaspersky Neuromorphic Platform usage examples.	This setting controls building a deb package with Kaspersky Neuromorphic Platform usage examples.
<code>KNP_BUILD_TESTS</code>	The following values are accepted: <code>ON</code>: enables test building. <code>OFF</code>: disables test building.	This setting controls test building.
<code>KNP_ENABLE_AVX</code>	The following values are accepted: <code>ON</code>: enables Advanced Vector Extensions. <code>OFF</code>: disables Advanced Vector Extensions.	This setting controls the use of Advanced Vector Extensions (avx).
<code>KNP_ENABLE_COVERAGE</code>	The following values are accepted: <code>ON</code>: enables testing of the platform's source code. <code>OFF</code>: disables testing of the platform's source code.	This setting controls whether the platform source code is covered by tests.
<code>KNP_IPO_ENABLED</code>	The following values are accepted: <code>ON</code> (default): enables optimization during linking. <code>OFF</code>: disables optimization during linking.	This setting controls whether optimization is used during linking.
<code>KNP_INSTALL</code>	The following values are accepted: <code>ON</code>: enables calls to the <code>install</code> command in CMake. <code>OFF</code>: disables calls to the <code>install</code> command in CMake.	This setting controls calling the <code>install</code> command in CMake.
<code>KNP_MAINTAINER_BUILD</code>	The following values are accepted: <code>ON</code>: enables the build for the maintenance specialist. <code>OFF</code>: disables the build for the maintenance specialist.	This setting controls the build for the maintenance specialist.

KNP_PYTHON_FRAMEWORK_BUILD	The following values are accepted: - ON: enables building the platform with the Python framework. This setting is automatically set to ON when stubgen is available on the device. - OFF: enables building the platform without the Python framework. If stubgen is not available on the device, the setting is automatically set to OFF.	This setting controls whether the platform will be built with the Python framework.
KNP_PYTHON_BUILD_WHEEL	The following values are accepted: - ON: enables building a whl package for the Python framework. - OFF: disables building a whl package for the Python framework.	This setting controls whether a whl package for the Python framework will be built. The setting depends on KNP_MAINTAINER_BUILD and KNP_PYTHON_FRAMEWORK_BUILD.

Presets for building the platform

The `CMakePresets.json` file, located in the root directory with the platform source code, describes the presets that you can use to [build the platform](#). The Kaspersky Neuromorphic Platform provides the following presets:

linux-default

Used as the default preset when building on Linux. Ninja is used as the build system.

windows-default

Used as the default preset when building on Windows. Visual Studio 17 2022 is used as the build system.

windows-nmake

Used for building on Windows with NMake.

windows-ninja

Used for building on Windows with Ninja.

Glossary

AltAI ANN2SNN backend

A backend for training and execution of artificial neural networks (ANN) converted to spiking neural networks running on an AltAI-2 neuromorphic processor.

AltAI SNN backend

A backend for execution of spiking neural networks running on an AltAI-2 neuromorphic processor.

Artificial Neural Network (ANN)

An artificial neural network that operates in layers, with numbers at the inputs and outputs of neurons of a layer.

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 ISI_{max} , where ISI_{max} 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 (IS_{max}) 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 spike or its absence.

Neuron function

Function that describes and modifies 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)

An artificial neural network that operates based on asynchronous transmission of impulses (spikes) between neurons.

Synapse

Neuron junction with the main function to transmit a signal from one neuron to another. Each neuron can be connected to other neurons via multiple synapses.

Synapse function

A function 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 neurons 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.

Winner Takes ALL (WTA) mechanism

A mechanism that is used in competitive training algorithms to select the most representative or dominant trait or pattern in a data set.

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_x86_64-<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-2 neuromorphic processor](#) and the [ANN2SNN](#) package.

Trademark notices

Registered trademarks and service marks are the property of their respective owners.

AMD and AMD64 are trademarks or registered trademarks of Advanced Micro Devices, Inc.

Apache is either a registered trademark or a trademark of the Apache Software Foundation.

Apple, Leopard, Mac, Mac OS, macOS, OS X, Objective-C, Safari, TrueType, and Xcode are trademarks of Apple Inc.

Arm is a registered trademark of Arm Limited (or its subsidiaries) in the US and/or elsewhere.

Borland is a trademark or registered trademark of Borland Software Corporation.

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.

FreeBSD is a registered trademark of The FreeBSD Foundation.

GITHUB is a trademark of GitHub, Inc., registered in the United States and other countries.

GITLAB is a trademark of GitLab Inc. in the United States and other countries and regions.

Google, Chrome, and Chromium are trademarks of Google LLC.

TensorFlow and any related marks are trademarks of Google LLC.

Intel, Core, and Pentium are trademarks of Intel Corporation or its subsidiaries.

IBM is a trademark of International Business Machines Corporation, registered in many jurisdictions worldwide.

Linux is the registered trademark of Linus Torvalds in the U.S. and other countries.

Microsoft, Consolas, Internet Explorer, OpenType, Segoe, Visual Studio, Win32, and Windows are trademarks of the Microsoft group of companies.

Firefox is a trademark of the Mozilla Foundation in the U.S. and other countries.

NVIDIA is a registered trademark of NVIDIA Corporation.

Java and JavaScript are registered trademarks 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.

QT is a trademark or registered trademark of The Qt Company Ltd.

UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.