DeepPhysX

Robin ENJALBERT, Alban ODOT, Stephane COTIN

Dec 15, 2022

PRESENTATION

1 Interfacing AI with simulation **1**

2 Let's get started **3**

2.1 About . . . 3
2.2 Overview . . . 5
2.3 Install . . . 9
2.4 CORE - Using a Pipeline . . . 11
2.5 CORE - Using an Environment . . . 14
2.6 CORE - Using a Network . . . 18
2.7 CORE - Using a Dataset . . . 21
2.8 CORE - Using a Visualizer . . . 22
2.9 CORE - Using the Stats . . . 23
2.10 CORE . . . 24
2.11 SOFA . . . 54
2.12 TORCH . . . 54

Index **55**

CHAPTER

ONE

INTERFACING AI WITH SIMULATION

The **DeepPhysX** project provides Python packages allowing users to easily interface their **numerical simulations** with **learning algorithms**.

DeepPhysX is mainly designed for SOFA and PyTorch frameworks, but other simulation and AI frameworks can also be used.

The project is closely linked to the SSD external Python library.

CHAPTER

TWO

LET'S GET STARTED

To better understand the architecture of **DeepPhysX**, reading the above sections is **highly recommended**:

- *About* → A quick summary of DeepPhysX features.
- *Overview* → The project architecture and packages.

Then, to start using **DeepPhysX**, please refer to the *Install* section to configure your own DeepPhysX installation. Some examples are provided in each `Example` repository to learn how to use the package, with a 3 leveled complexity:

- **Tutorial**: A walkthrough of **DeepPhysX** to learn the basics.
- **Features**: A full session reviewing all features of **DeepPhysX** via a simple example.
- **Demos**: Demonstration sessions with high-level applications.

Finally, to go further in understanding some components of **DeepPhysX**, refer to the associated section.

2.1 About

2.1.1 Features

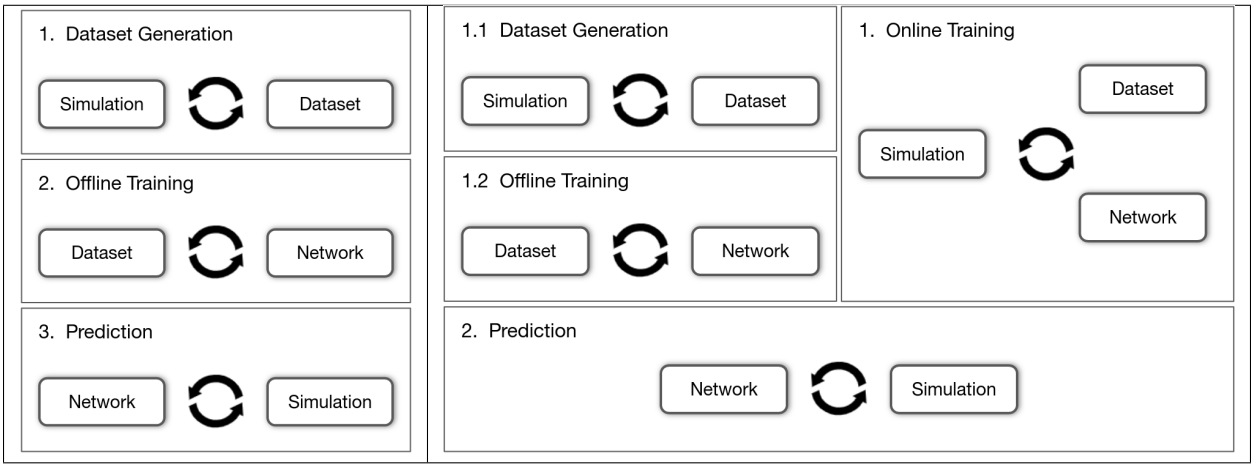
The purpose of the **DeepPhysX** framework is to provide an interface between **Deep Learning algorithms** and **numerical simulations**.

This **full Python3** project brings several **pipelines**, allowing the user to:

- **Generate a dataset** with synthetic data from numerical simulations;
- **Train an artificial neural network** with a synthetic dataset;
- **Use the predictions of a trained network** inside a numerical simulation.

Note: The **dataset generation** and the **training** pipelines can be done **simultaneously**.

Table 1: Classical pipelines (left), DeepPhysX pipelines (right)



About **dataset** components, the project has the following features:

- Automatic training dataset **storage** and **loading** with **multiple files management**;
- Dataset **shuffle** and **normalization**;
- Multiple dataset **modes**: Training, Validation, Prediction;
- **Customizable** dataset fields.

About **simulation** components, the project has the following features:

- **Data generation** achieved by several simulations running in **multiprocessing** with a client-server architecture;
- Operation with **internal data**, from the **dataset** or from the **neural network**;
- Increased **interactions** with other components (dataset, neural network, visualizer);
- Check the **validity** of the training **data**.

About **neural network** components, the project has the following features:

- Automatic **storage** and **loading** of networks during training;
- **Customizable data transformations** at each step (forward pass, optimization, prediction apply);
- **Customizable optimization process** with training data;
- Already **implemented architectures**: FC, UNet.

Additional tools are also provided:

- A **visualization** Factory to **init**, **update** and **render** the simulated objects (written with Vedo);
- An customizable analysis of the **evolution** of the **training session** (written with Tensorboard).

2.1.2 Working sessions

A **DeepPhysX** pipeline is associated with a **working session**. The repository is **automatically managed** and typically contains the following **tree structure**:

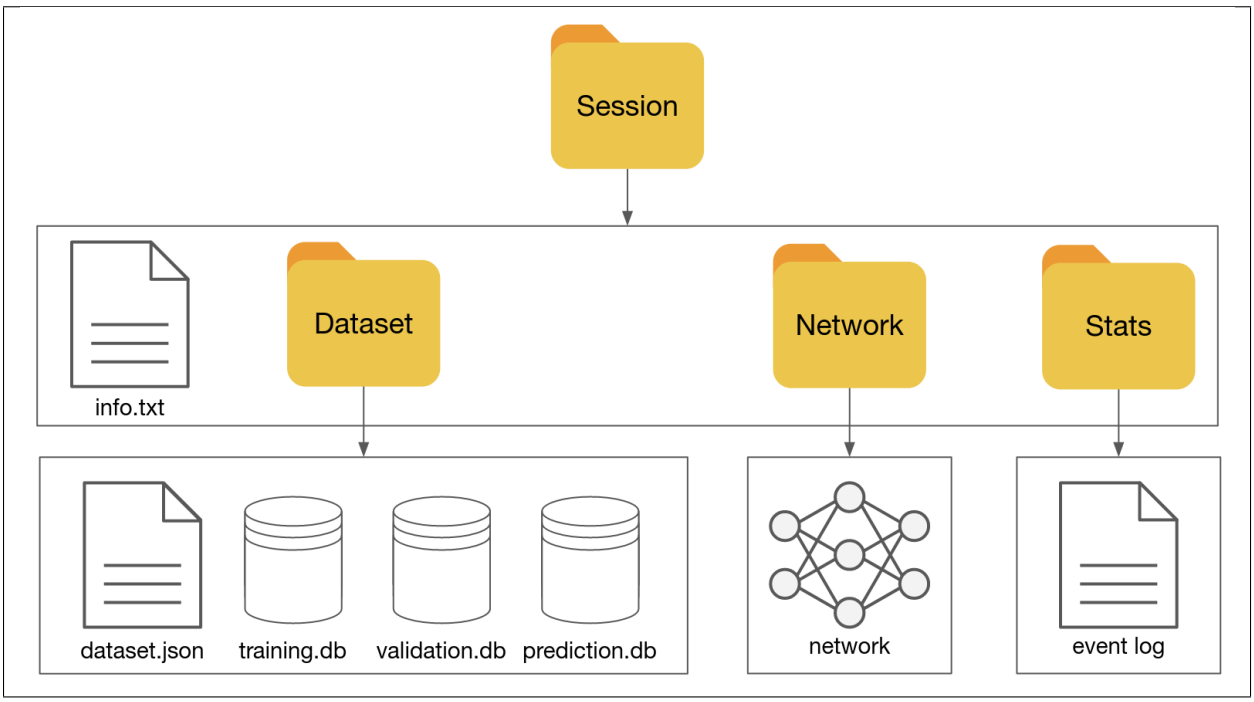
`Dataset`
: Contains the **dataset partitions** and a **json description file**;

`Network`
: Contains the saved **neural networks**;

`Stats`
: Contains a **Tensorboard file** which gathers **training analysis**;

`info.txt`
: A **text description file** which gathers the **configurations** of all the **involved components**.

Table 2: Working session repository



2.2 Overview

2.2.1 Packages

The **DeepPhysX** project is divided into several **Python packages**: a CORE package which is able to communicate with a SIMULATION package and an AI package. This way, the CORE has no dependencies neither to a SIMULATION framework nor an AI framework, and any of those frameworks can be compatible with **DeepPhysX**.

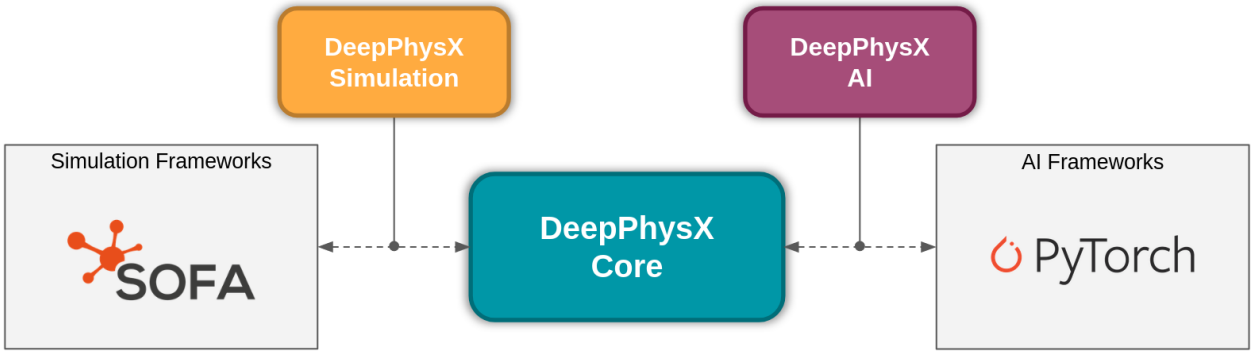


Fig. 1: Package organization

Core

This package rules the **data flow** and the **communication** between the AI components and the simulation components. Data flow also involves the **storage** and **loading** of datasets and neural networks, the **management** of the visualization tool and the **analysis** of the training sessions. This package is named *DeepPhysX.Core*.

Dependencies

NumPy, Tensorboard, Vedo, SimulationSimpleDatabase

Simulation

A **simulation package** provides a **DeepPhysX** compatible API for a **specific simulation framework**. For **DeepPhysX**, each simulation package dedicated to a specific simulation framework (written in python or providing python bindings). Thus, each package will be named according to the framework with a common prefix, for instance *DeepPhysX.Simulation*.

Available simulation packages

DeepPhysX.Sofa designed for SOFA

AI

An **AI package** provides a **DeepPhysX** compatible API for a **specific AI framework**. In the same way, each learning package is dedicated to a specific AI Python framework. Thus, each package will be named according to the framework with the same common prefix, for instance *DeepPhysX.AI*.

Available learning packages

DeepPhysX.Torch designed for PyTorch

2.2.2 Architecture

This section describes both the links between the components of CORE package and the links to AI and SIMULATION packages.

Users might use one of the provided *Pipelines* for their **data generation**, their **training session** or their **prediction session**. These *Pipelines* trigger a **loop** which defines the number of samples to produce, the number of epochs to perform during a training session or the number of steps of prediction.

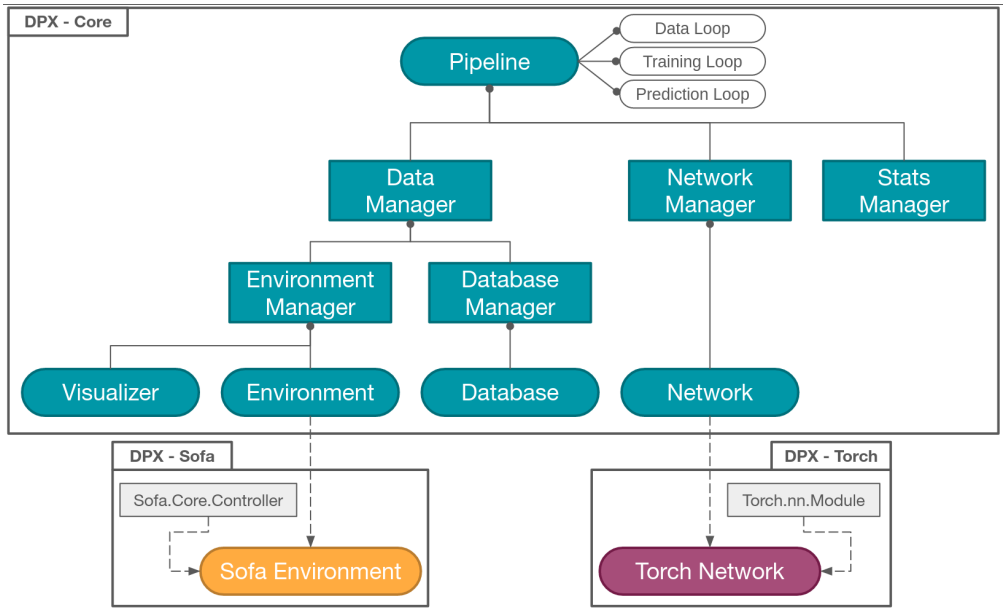


Fig. 2: Components architecture

The *Pipeline* will involve several components (data producers and data consumers), but the *Pipeline* will always communicate with their *Manager* first. A main *Manager* will provide the *Pipeline* an intermediary with all the existing *Managers*:

`DatabaseManager`
: It will manage the *Database* component to create **storage** partitions, to fill these partitions with the synthetic training data produced by the *Environment* and to **reload** an existing *Database* for training or prediction sessions.

> **Note:** If training and data generation are done simultaneously (by default for the training *Pipeline*), the *Database* can be built only during the first epoch and then reloaded for the remaining epochs.

`EnvironmentManager`
: It will manage the *Environment* (the numerical simulation) component to **create** it, to trigger **steps** of simulations, to **produce** synthetic training data, to provide **predictions** of the network if required, and to finally **shutdown** the *Environment*.

> **Note:** This *Manager* can communicate directly with a single *Environment* or with a *Server* which shares information with several *Environments* in multiprocessing launched as *Clients* through a custom TCP-IP protocol (see *dedicated section*).

Note: The two above *Managers* are managed by the `DataManager` since both the *Environment* and the *Database* components provide training data to the *Network*. This `DataManager` is the one who decides if data should be requested from the *Environment* or from the *Database* depending on the current state of the *Pipeline* and on the components configurations.

`NetworkManager`

It will manage several objects to **train** or **exploit** your *Network*:

- The *Network* to produce a **prediction** from an input, to **save** a set of parameters or to **reload** one.
- The *Optimizer* to compute the **loss** value and to **optimize** the parameters of the *Network*. This component uses existing loss functions and optimizers in the chosen AI framework.
- The *DataTransformation* to **convert** the type of training data sent from *Environment* to a compatible type for the AI framework you use and vice versa, to **transform** training data before a prediction, before the loss computation and before sending the prediction to the *Environment*.

Note: The above components are designed to be easily inherited and upgradable if the content of AI packages is not enough. Users are thus free to define their own *Network* architecture, to create a custom loss or optimizer to feed the *Optimizer* and to compute the required tensor transformations in the *DataTransformation*.

`StatsManager`

It will manage the **analysis** of the **evolution** of a training session. These analytical data will be saved in an **event log file** interpreted by Tensorboard.

Note: Usual curves will be automatically provided in the board (such as the evolution of the loss value, the smoothed mean and the variance of this loss value per batch and per epoch), but other custom fields can be added and filled as well.

Warning: It is not possible to use the default *Network* and *Environment* provided in the CORE package, since they are not implemented at all. The reason is that you need to choose an AI and a SIMULATION Python framework to implement them. The aim of **DeepPhysX** additional packages is to provide a compatible implementation both for **DeepPhysX** and these frameworks.

Example

If you choose PyTorch as your AI framework, you can use or implement a *TorchNetwork* which inherits from both the CORE *Network* and `Torch.nn.module` (available in *DeepPhysX.Torch*).

If you choose SOFA as your SIMULATION framework, you can implement a *SofaEnvironment* which inherits from both the CORE *Environment* and `Sofa.Core.Controller` (available in *DeepPhysX.Sofa*).

2.3 Install

It is recommended to read the *Packages* section first to understand the packages organisation.

The CORE package *DeepPhysX.Core* is always installed by default.
Using a SIMULATION package is recommended but not required.
At least one AI package is mandatory (default one is *DeepPhysX.Torch*).

2.3.1 Prerequisites

DeepPhysX packages have the following dependencies :

Package	Dependency	Type	Install
DeepPhysX_Core	Numpy	**Required**	`pip install numpy`
	Vedo	**Required**	`pip install vedo`
	Tensorboard	**Required**	`pip install tensorboard`
	SSD	**Required**	`pip install SimulationSimpleDatabase`
DeepPhysX_Sofa	SOFA Framework	**Required**	Follow instructions
	SofaPython3	**Required**	Follow instructions
	Caribou	Optional	Follow instructions
DeepPhysX_Torch	PyTorch	**Required**	`pip install torch`

Note: *DeepPhysX.Sofa* has a dependency to Caribou to run the demo scripts from `Examples/SOFA/Demo` since implemented simulations involve some of its components.

2.3.2 Install

Install with pip

DeepPhysX packages are all registered on PyPi. They can easily be installed with `pip`:

```
$ pip3 install DeepPhysX
$ pip3 install DeepPhysX.Sofa
$ pip3 install DeepPhysX.Torch
```

Then, you should be able to run:

```
$ pip3 show DeepPhysX
$ pip3 show DeepPhysX.Sofa
$ pip3 show DeepPhysX.Torch
```

```
import DeepPhysX.Core
import DeepPhysX.Sofa
import DeepPhysX.Torch
```

Install from sources

DeepPhysX packages must have the following architecture. These detailed instructions will build such an installation:

```
DeepPhysX
 ├── Core
 ├── Sofa
 └── Torch
```

Start by cloning the **DeepPhysX** source code from its Github repository in a directory named `DeepPhysX/Core`:

```
$ mkdir DeepPhysX
$ cd DeepPhysX
$ git clone https://github.com/mimesis-inria/DeepPhysX.git Core
```

Then, you can add compatibility layers to your **DeepPhysX** environment and install packages:

- **Option 1 (recommended):** run one of the `setup_.py` scripts that handle the installation of all packages
  - Use `setup_user.py` to install and manage packages with `pip` as non-editable.

    ```
    $ python3 setup_user.py
    ```

  - Use `setup_dev.py` to link packages in the site-packages.

    ```
    $ python3 setup_dev.py set
    ```

  Note: Both scripts will asks the packages to install and will **automatically clone** missing packages.

- **Option 2:** clone the corresponding Github repositories in the created `DeepPhysX` directory, then install packages manually.

  ```
  # Clone compatibility layers
  $ git clone https://github.com/mimesis-inria/DeepPhysX.Sofa.git Sofa
  $ git clone https://github.com/mimesis-inria/DeepPhysX.Torch.git Torch

  # Install packages manually
  $ cd Core ; pip3 install .
  $ cd ../Sofa ; pip3 install .
  $ cd ../Torch ; pip3 install .
  ```

Finally, you should be able to run:

```
# If installed with pip
$ pip3 show DeepPhysX
$ pip3 show DeepPhysX.Sofa
$ pip3 show DeepPhysX.Torch
```

```
# In both cases
import DeepPhysX.Core
import DeepPhysX.Sofa
import DeepPhysX.Torch
```

2.4 CORE - Using a Pipeline

Dedicated code in *Core/Pipelines* module.

2.4.1 General Policy

Several *Pipelines* are available with **DeepPhysX**, allowing the user to:

- **Generate** synthetic data from simulations → `DataGeneration`
- **Train** artificial neural networks with synthetic data → `Training`
- Use the **predictions** of trained networks inside a simulation → `Prediction`

A *Pipeline* is always associated with a *working session*, whether it already exists or whether it is automatically created when the *Pipeline* is launched.

The main policy for creating and using a *Pipeline* is to first define a *Configuration* for each component involved by this *Pipeline*. *Configurations* can then be mandatory for *Dataset*, *Network* and *Environment* if one wants to create a *Pipeline*. Once these *Configurations* are defined, the *Pipeline* can be created and launched.

Note: More details are provided for each *Configuration* in dedicated sections.

2.4.2 Pipeline - Data generation

The *DataGeneration* will only involve an *Environment* and a *Dataset*, so this *Pipeline* requires the corresponding *Configurations*.

As the purpose of this *Pipeline* is only to create synthetic data, the working session will always be created at the same time. The name of the repository to create can be provided as a parameter.

Furthermore, users have to define which data to save and how much :

- The number of **batches** to produce
- The number of **samples** in a single batch

Warning: This pipeline does not involve a *Network*, the `get_prediction` request from Environment is then disabled. Trying to launch this request it will lead to an error, so make sure there are no such requests in the data production procedure in your *Environment*.

See following example:

```
# Import BaseDataGeneration and Config objects
from DeepPhysX.Core.Pipelines.BaseDataGeneration import BaseDataGeneration
...

# Define configs
dataset_config = ...
environment_config = ...

# Create the pipeline
data_generator = BaseDataGeneration(session_dir='sessions',
```

(continues on next page)

(continued from previous page)

```
                                        session_name='my_data_generation',
                                        dataset_config=dataset_config,
                                        environment_config=environment_config,
                                        batch_nb=500,
                                        batch_size=16)

# Launch the pipeline
data_generator.execute()
```

2.4.3 Pipeline - Training

The *Training* can involve an *Environment*, a *Dataset* and a *Network*, so this *Pipeline* might require the corresponding *Configurations*. There are several ways to use this pipeline:

Training a Network from scratch
To train a *Network* from scratch, the *Training* requires the whole set of *Configurations*. A new working session will be created, whose name can be set as a parameter.

Training a Network with an existing Dataset
Training a new *Network* with an existing *Dataset* is considered as creating a new working session. The path to the *Dataset* to use has to be provided as a parameter. Using an *Environment* is not mandatory since the training data can already have the right format to feed the *Network*. If some data computation must be performed between the *Dataset* and the *Network*, an *Environment* can be created with the specific *Configuration* (see more in the dedicated section).

Training a Network from an existing Network state
Training from an existing *Network* state can be done both in an existing session or in a new session. If you want to work in the same session, you have to configure the *Training* to do so, otherwise a new working session will be automatically created. In the same session, a new set of trained parameters will be added in the `network` repository, either trained with data from an external *Dataset* (whose path must be provided) or with data from the *Environment* (whose *Configuration* must be provided).

The last parameters to set in the *Trainer* are:

- The number of **epochs** the training loop should complete during the session
- The number of **batches** used during a single epoch
- The number of **samples** in a single batch

Note: By default, the training data will be produced inside the *Environment* during the first epoch and then re-used from the *Dataset* for the remaining epochs. If you always need to use data from the *Environment*, you can specify this in its *Configuration*.

See following example:

```
# Import BaseTraining and Config objects
from DeepPhysX.Core.Pipelines.BaseTraining import BaseTraining
...

# Define configs
dataset_config = ...
environment_config = ...
```

(continues on next page)

(continued from previous page)

```
network_config = ...

# Create the pipeline
trainer = BaseTraining(session_dir='sessions',
                       session_name='my_training',
                       dataset_config=dataset_config,
                       environment_config=env_config,
                       network_config=net_config,
                       epoch_nb=100,
                       batch_nb=500,
                       batch_size=16)

# Launch the pipeline
trainer.execute()
```

2.4.4 Pipeline - Prediction

The *Prediction* always requires a *Network* to compute predictions and an *Environment* to apply them, so this *Pipeline* will always require the corresponding *Configurations*.

This *Pipeline* always works with an existing working session, no new sessions can be created within a *Prediction*. The path to the session is therefore required, assuming that it contains a trained *Network*.

The *Prediction* can either run a specified **number of steps** or run an **infinite loop**.

A *Dataset* configuration can be provided. In this case, the *Prediction* can record prediction data. Each sample computed during the prediction phase will then be added to the *Dataset* in dedicated partitions. With a *Dataset*, the *Prediction* can also load its data to **replay** stored samples.

See following example:

```
# Import BasePrediction and Config objects
from DeepPhysX.Core.Pipelines.BasePrediction import BasePrediction
...

# Define configs
dataset_config = ...
environment_config = ...
network_config = ...

# Create the pipeline
runner = BasePrediction(session_dir='sessions',
                        session_name='my_training',
                        dataset_config=dataset_config,
                        environment_config=env_config,
                        network_config=net_config,
                        step_nb=-1)

# Launch the pipeline
runner.execute()
```

2.5 CORE - Using an Environment

Dedicated code in *Core/Environment* and *Core/AsyncSocket* modules.

2.5.1 Implementation

An *Environment* can be considered either as a *TcpIpClient* (communication with the *EnvironmentManager* through a *TcpIpServer*, see *dedicated section* below) or not (direct communication with the *EnvironmentManager*). Yet, the user API will be the same if the *Environment* is a *TcpIpClient* entity or not, so that the user can implement its *Environment* regardless of this dependence.

Mandatory methods

The following methods must be implemented.

`create`	Create the simulation. This method is automatically called when the *Environment* component is created.
`init_database`	Define the fields of the training dataset. This method is automatically called when the *Environment* component is created.
`step`	Describe the transitions between simulation states. This method is automatically called when the *Pipelines* require a step in the simulation to produce a data sample.

Optional methods

Additional non mandatory methods can be implemented to describe the simulation behavior.

`init`	Initialize the simulation. In some cases, the simulation components might need to be initialized in addition to being created. This method is automatically called when the *Environment* component is created, right after the create method.
`init_visualization`	Define the visualization objects to send to the *Visualizer*. A *Factory* is available in the *Environment* to easily design visualization objects. This method is automatically called when the *Environment* component is created, right after the init.
`check_sample`	Each step will be followed by a sample checking. This check returns True by default but samples can be sorted and then not added to the *Dataset* if they do not meet some criteria.
`apply_prediction`	In some use cases, the *Network* predictions can be sent back to the *Environment* to be applied in the simulation. This method describes how the prediction must be used in the simulation.
`close`	For some reasons, the simulation might need to perform some tasks before being shut down. These tasks can be done in the close method which is automatically called before shut down.

Data producing methods

The following methods are dedicated to data producing /exchange.

`define_training_data`	This method must be used in `init_database` to specify the training data fields names and types.
`define_additional_data`	This method must be used in `init_database` to specify the additional data fields names and types.
`set_training_data`	This method defines which data will be considered as training data (data sent to the *NetworkManager* to feed the *Network*).
`set_additional_data`	In addition to the training data, some additional data can be sent directly to the dataset to replay some simulation states. This method adds a new field to the *Dataset* and must be then called at each step.

Requests

The *Environment* is also able to perform some requests. These requests are sent either directly to the *EnvironmentManager* or through a *TcpIpServer*.

`save_parameters`	Save a set of parameters in the Database.
`load_parameters`	Load a set of parameters from the Database.
`get_prediction`	Depending on the application, a prediction of the *Network* can be useful before the end of a step. By default, a *Network* computes predictions after a simulation step, so the only way to get one during a step is by this request. The training data must obviously be set before triggering this request.
`update_visualization`	If a *Visualizer* was created, it must be manually updated at each step by sending the updated state of the simulation. The factory allows to easily create updated visualization data from objects ids with `update_obj` methods. User only has to fill the object id (same as initialization order) and the required fields (detailed for each object in *dedicated section*).

2.5.2 Configuration

Using an *Environment* in one of the **DeepPhysX** *Pipeline* always requires an *EnvironmentConfig*. The role of this component is both to bring together all the options for configuring an *Environment* and to create an instance of a single *Environment* or launch a *TcpIpServer* with several *TcpIpClients*. In the first case, the single *Environment* will simply be created within the `create_environment` method, while in the other case, the `create_server` method will simply create and launch a *TcpIpServer* and then start several subprocesses, each using the `launcherBaseEnvironment.py` script to create and launch an *Environment* as a *Client*.

Simulation parameters

Here is a description of attributes related to a base *Configuration*.

environment_class	The class from which an *Environment* will be created. The attribute requires the class and not an instance, as it will be automatically created as explained above.
simulations_per_step	The number of iterations to compute in the *Environment* at each time step. An *Environment* will compute one iteration by default.
visualizer	A visualization tool is provided, which renders the specified parts of each *Environment*. If no *Visualizer* is provided, the pipeline will run without any render window.
env_kwargs	*Environments* can receive additional parameters within this dictionary if they need to be parameterized differently.

Data parameters

Here is a description of attributes related to sample generation.

always_produce	This flag is useful for the training *Pipeline*. If False (by default), the *DataManager* requests batches to the *Environment* during the first epoch and reload them next. If set to True, the *DataManager* requests new batches to the *Environment* during the whole training session.
load_samples	If True, the dataset will always be used in the environment.
only_first_epoch	If True, data will always be created from *Environment*. If False, data will be created from the *Environment* during the first epoch and then re-used from the *Dataset*.

TcpIP parameters

Here is a description of attributes related to the *Client* configuration.

as_tcp_ip_client	Defines if the *Environment* will be launched as a *Client* (True by default) or directly connected to its *EnvironmentManager*. Note: In the prediction *Pipeline*, only one *Environment* will be used, so the value of this flag is ignored.
ip_address	Name of the IP address to bind *TcpIpObjects*. The default value is "localhost" to host the *Server* and *Clients* locally.
number_of_thread	The number of *Environments* to launch simultaneously if the flag `as_tcp_ip_client` is True.
port	TCP port's number through which *TcpIpObjects* will communicate (10000 by default).

See following example:

```
# Import EnvironmentConfig and Visualizer
from DeepPhysX_Core.Environment.BaseEnvironmentConfig import BaseEnvironmentConfig
from DeepPhysX_Core.Visualizer.VedoVisualizer import VedoVisualizer

# Create the config
env_config = BaseEnvironmentConfig(environment_class=MyEnvironment,
                                   visualizer=VedoVisualizer,
                                   as_tcp_ip_client=True,
                                   number_of_thread=5,
```

(continues on next page)

(continued from previous page)

```
                                        ip_address='localhost',
                                        port=10000)
```

2.5.3 Client-Server Architecture

DeepPhysX allows users to run several *Environments* in **multiprocessing** to speed up data generation. The *Core/AsyncSocket* module defines a **Client-Server architecture** where a *TcpIpServer* communicates with several *TcpIpClients* using a **TcpIp protocol**.

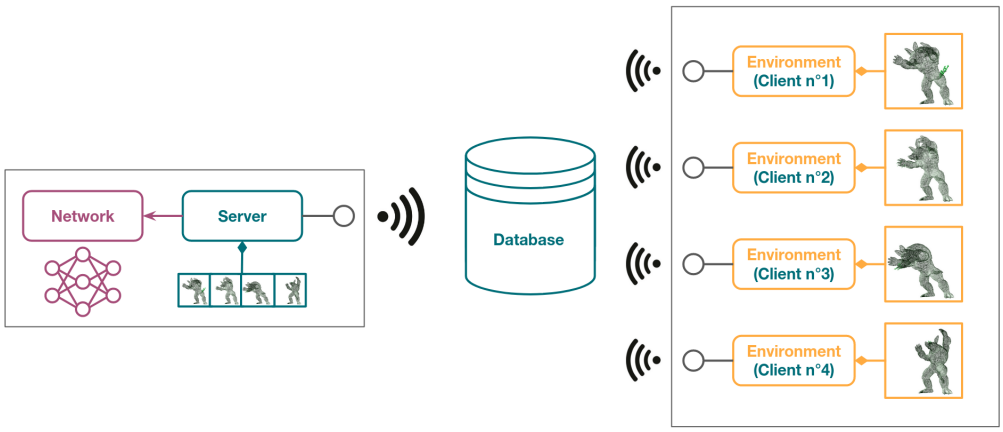


Fig. 3: Client-Server Architecture

At the beginning of data generation, a *TcpIpServer* is launched by the *EnvironmentConfig*. This *TcpIpServer* binds to the socket with the configured IP address on the configured port and then listens for *TcpIpClients*. To launch *TcpIpClients*, the *EnvironmentConfig* runs sub-processes where a launcher is called. This launcher creates a *TcpIpClient* instance, this *TcpIpClient* is then bound to the socket with the configured IP address on the configured port. Once the *TcpIpClients* are all connected to the *TcpIpServer*, initialization is performed to create all the *Environments* and initialize the parameters exchanges. *TcpIpClients* are now ready to communicate with the *TcpIpServer* to handle the data generation. Finally, the *TcpIpServer* triggers the shutdown of each *TcpIpClient* and closes the socket.

Both *TcpIpServer* and *TcpIpClient* inherit from *TcpIpObject* to access low levels of sending and receiving data on the socket. The data is sent as a custom bytes message converted with a *BytesConverter*, which handles Python common types and NumPy arrays. On top of these low level data exchange methods are built higher level protocols to send labeled data, labeled dictionaries and commands.

2.6 CORE - Using a Network

Dedicated code in *Core/Network* module.

2.6.1 Network Implementation

A *Network* cannot be directly implemented from *BaseNetwork* because there is no dependency on any Python AI package. Even if a *BaseNetwork* is not usable, any **DeepPhysX** AI package provides the same API.

Note: Before writing code to create a new *Network* architecture, please check **DeepPhysX** AI packages if it is not already implemented.

Specific methods

The following methods are the only methods requiring a user implementation.

`__init__`	Define the *Network* architecture.
`predict`	Define the data fields used to compute a prediction.
`forward`	Define the forward pass of the *Network*.

Shared methods

The following methods share the same implementation within the same AI package, regardless of *Network* architecture. Thus, they should already be implemented.

`set_train`	Configure the *Network* to training mode (chain rule for gradient will be calculated).
`set_eval`	Configure the *Network* to prediction mode (no gradient will be calculated).
`set_device`	Define whether tensors computations are done on the CPU or on the GPU.
`load_parameters`	Load a set of saved parameters from a file. This set of parameters must be compatible with the current *Network* architecture.
`get_parameters`	Return the current state of *Network* parameters.
`save_parameters`	Save the current set of parameters to a file..
`nb_parameters`	Return the number of parameters in the architecture.
`numpy_to_tensor`	Convert a Numpy array to a tensor with the compatible type. Received data from Core will always be Numpy arrays.
`tensor_to_numpy`	Convert a tensor to a Numpy array. Data provided to Core must be converted to Numpy arrays.

Data fields

The training data is given to the *Network* as a dictionary of tensors. By default, the data used for inference has a single "input" field, the data produced by the *Network* has a single "prediction" field, the data used for optimization has a single "ground_truth" field. If another field should be defined, the *Network* class should specify them in the corresponding `self.net_fields`, `self.pred_fields` & `self.opt_fields` variables.

2.6.2 Optimization Implementation

The purpose of the *Optimization* component is to compute the loss associated with the prediction of the *Network* and to optimize its parameters considering the gradient. Same as *BaseNetwork*, *BaseOptimization* cannot be used in training sessions as it must depend on a Python AI framework. However, *Optimization* components still share the same API in every DeepPhysX package. Given a Python AI framework, methods should all be already implemented so that users can use them as it is.

Init methods

The first methods allow defining the loss function and the optimizer. These methods should not be overwritten by users.

`set_loss`	Initialize the loss function.
`set_optimizer`	Initialize the *Optimizer* with the parameters of the *Network*.

Optimization methods

Other methods define the loss computation and the optimization process. Users can use the default implementation or customize their own processes.

`compute_loss`	Compute the loss with the defined loss function from prediction and ground truth.
`transform_loss`	Apply a transformation on the loss value using potential additional data. Additional loss data can be sent from *Environment* (see *dedicated section*).
`optimize`	Run an optimization step to adapt the parameters of the *Network* according to the loss gradient.

2.6.3 Data Transformation Implementation

DataTransformation objects are dedicated to tensor transformations at different levels when streaming data through *Network* and *Optimization* objects. These transformations are automatically called with the pipeline and apply the identity transformation by default. Users are then free to define their own tensor transformations with the following methods.

`transform_before_prediction`	Apply a tensor transformation to the *Network* data (before a prediction).
`transform_before_loss`	Apply a tensor transformation to the ground truth data and / or the prediction data (after the prediction, before the loss computation).
`transform_before_apply`	Apply a tensor transformation to the prediction data (before sending it to the *Environment*).

2.6.4 Configurations

Using a *Network*, an *Optimizer* and a *DataTransformation* in one of the DeepPhysX *Pipeline* always requires a *NetworkConfig*. This *Configuration* object role is both to bring together all the options for configuring this set of objects and to create them. Objects are created within the `create_network`, `create_optimization` and `create_data_transformation` methods respectively.

Network parameters

Here is a description of attributes related to *Network* configuration.

`network_class`	*Network* class from which an instance will be created.
`network_dir`	Path to an existing *Network* repository with saved parameters.
`network_name`	Name of the *Network* object.
`network_type`	Type of the *Network*.
`which_network`	If several sets of parameters are saved in `network_dir`, load the specified one. By default, the last save is loaded.
`save_each_epoch`	If True, *Network* parameters will be saved at the end of each training epoch. Otherwise, they are only saved at the end of the training session.

Optimization parameters

Here is a description of attributes related to *Optimization* configuration.

`optimization_class`	*Optimization* class from which an instance will be created.
`lr`	Learning rate value.
`loss`	Loss class that will be used to compute loss value.
`optimizer`	*Optimizer* class that will be used to optimize the *Network* parameters.
`require_training_stuff`	In cases where a loss class and / or an optimizer class (training stuff) are not used for training, users must set this option to False.

DataTransformation parameters

Here is a description of attributes related to *DataTransformation* configuration.

`data_transformation_class`	*DataTransformation* class from which an instance will be created.

See following example:

```
# Import NetworkConfig
from DeepPhysX_Core.Dataset.BaseNetworkConfig import BaseNetworkConfig

# Choose classes
MyNetwork = ...              # Define a network architecture or take one from an AI␣
↪package
MyOptimizationClass = ...    # Pick the one from an AI package
```

(continues on next page)

(continued from previous page)

```
MyDataTransformation = ...  # Define tensor transformations or use default one from an␣
↪AI package
MyLoss = ...                # Choose one from the AI base framework
MyOptimizer = ...           # Choose one from the AI base framework

# Create the config
network_config = BaseNetworkConfig(network_class=MyNetwork,
                                   optimization_class=MyOptimizationClass,
                                   data_transformation_class=MyDataTransformation,
                                   lr=1e-5,
                                   loss=
                                   loss=MyLoss,
                                   optimizer=MyOptimizer)
```

2.7 CORE - Using a Dataset

Dedicated code in *Core/Dataset* module.

2.7.1 Behavior

DeepPhysX comes with its own *Dataset* management system, using the features from the SSD library. The synthetic data produced in *Environments* is stored on a *Database* that is in a dedicated repository named `dataset` in the training session. Data is stored as partitions: these partitions correspond to the different *Dataset* modes (training data, test data, prediction data) and can be multiple for each data field in order not to exceed the maximum size of the current amount of loaded data. Each partition will have a unique name: `<session_name>_<dataset_mode>_<field>_<index>.npy`.

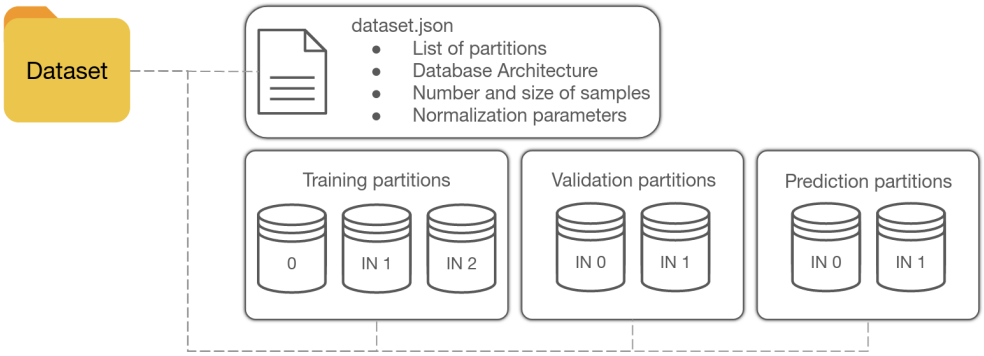


Fig. 4: Dataset repository organisation.

When adding a batch to the *Dataset*, a new partition is created for each data field if the current *Dataset* size exceeds the threshold. The batch is then appended to the *Dataset* for each data field. Default *Dataset* fields are inputs and outputs, but users can define any data field from *Environment* (see *dedicated section*). Each field must always be filled at each batch. A `dataset.json` file gathers information about the produced dataset and normalization coefficients if the normalization is applied.

When loading data from an existing *Dataset*, the partitions are loaded and can be accessed randomly or not among the whole set of partitions.

2.7.2 Configuration

A *Dataset* is almost always used in DeepPhysX pipelines and requires a *DatasetConfig*. This *Configuration* object role is to bring together all the options to configure the *Dataset* and to create it with `create_dataset` method.

Here is a description of attributes related to *Dataset* configuration.

`existing_dir`	Path to an existing *Dataset* repository if this repository needs to be loaded or completed.
`max_file_size`	Maximum size (in Gb) of the total *Dataset* object.
`mode`	Specify the *Dataset* mode between "Training", "Validation" and "Running".
`normalize`	If True, normalization parameters are computed from training data and applied to any loaded data.
`shuffle`	Specify if the loading order is random or not (True by default).

See following example:

```
# Import DatasetConfig
from DeepPhysX_Core.Database.BaseDatabaseConfig import BaseDatabaseConfig

# Create the config
database_config = BaseDatabaseConfig(max_file_size=1,
                                     shuffle=True,
                                     mode='Training',
                                     normalize=True)
```

2.8 CORE - Using a Visualizer

Dedicated code in *Core/Visualizer* module.

2.8.1 How to use

DeepPhysX provides a visualization tool written with Vedo (a Python library based on Numpy and VTK) called *VedoVisualizer*, using the implementation form the SSD library. This *Visualizer* brings several advantages:

- Users can add any component of the simulation in the *Visualizer*;
- This *Visualizer* is compatible with every *Pipeline*;
- Parallel running *Environments* are rendered in the same window with sub-windows;
- A *Factory* is created with each *Environment* so that users can access templates to define visualization data.

Objects are created using the `add_<object>` methods of the *Factory* in the *Environment*. These methods require the fields detailed above (common fields and object-specific fields). These objects must be added in the `init_visualization` method of the *Environment*, which is automatically called to create the objects within the *Factory*.

Objects are updated using the `update_<object>` methods of the *Factory* in the *Environment*. These methods require the object index (indices follow the order of creation) and a the updated data fields. The *Factory* will only use templates to create an updated objects dictionary which must be sent with the request `update_visualization` to update the view.

Visual Objects Parameters

All the available objects and their parameters are listed on the SSD documentation.

2.8.2 Configuration

Configuring a *Visualizer* is very simple, since the only option to change is the `visualizer` field in the *EnvironmentConfig*. If set to None, no *Visualizer* will be created, even if the *Environment* uses its *Factory* to create and update visualization data. It must be set to *VedoVisualizer* to activate the visualization tool.

See following example:

```
# Import EnvironmentConfig and Visualizer
from DeepPhysX_Core.Environment.BaseEnvironmentConfig import BaseEnvironmentConfig
from DeepPhysX_Core.Visualization.VedoVisualizer import VedoVisualizer

# Create the config
env_config = BaseEnvironmentConfig(environment_class=MyEnvironment,
                                   visualizer=VedoVisualizer)
```

2.9 CORE - Using the Stats

2.9.1 How to Use

During the training session, some optimizations analysis are performed and stored as a readable file for Tensorboard. This file is automatically filled at each *Network* optimization step in the `stats` repository of the current working directory.

Tensorboard will be automatically launched for the current training session in a web browser. To open a previous training session analysis, launch Tensorboard with:

```
$ cd <working_session>/stats
$ tensorboard --logdir .
>> Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --
↪bind_all
   TensorBoard x.x.x at http://localhost:6006/ (Press CTRL+C to quit)
```

Then copy-paste the provided link into a web browser.

2.9.2 Adding fields

By default, six data fields will be stored:

- The **value** of the **loss** function at each batch / epoch
- The **mean** of the **loss** function at each batch / epoch over the last 50 elements
- The **variance** of the **loss** function at each batch / epoch

Custom data fields can be set in Tensorboard as well. After an optimization step, the *Optimization* class will by default return a dictionary with a single item named `loss` containing the current loss value. Then, the *Trainer* will add every item contained in this dictionary to the *StatsManager*. To fill this dictionary with custom fields, a custom *Optimization* class must be implemented where the `transform_loss` method will fill that dictionary (see *dedicated section*). This new field will have the same name as the item and take each value given in this item at each optimization step.

See following example:

```
# Import BaseOptimization
from DeepPhysX_Core.Network.BaseOptimization import BaseOptimization

# Create a new Optimization class
class MyOptimization(BaseOptimization):

    # Override the transform_loss method
    def transform_loss(self, data):
        return {'loss': self.loss_value     # Default loss value
                'new_field': ...}           # Additional field
```

2.10 CORE

AsyncSocket	*Dataset*	*Environment*	*Manager*
BytesConverter *TcpIpClient* *TcpIpObject* *TcpIpServer*	*DatabaseHandler* *BaseDatabaseConfig*	*BaseEnvironment* *BaseEnvironmentConfig*	*DataManager* *DatabaseManager* *EnvironmentManager* *NetworkManager* *StatsManager*

Network	*Pipelines*	*Visualizer*
BaseNetwork *BaseNetworkConfig* *BaseOptimization* *DataTransformation*	*BaseDataGeneration* *BasePipeline* *BasePrediction* *BaseTrainer*	*VedoFactory* *VedoVisualizer*

2.10.1 AsyncSocket

BytesConverter

class BytesConverter.**BytesConverter**

bytes_to_data(*bytes_fields: List[bytes]*) → Union[None, bytes, str, bool, int, float, List, ndarray]

Recover data from bytes fields. Available types: None, bytes, str, bool, signed int, float, list, ndarray.

Parameters
bytes_fields – Bytes fields (Type, Data, Args).

Returns
Converted data.

data_to_bytes(*data: Union[None, bytes, str, bool, int, float, List, ndarray], as_list: bool = False*) → Union[bytes, List[bytes]]

Convert data to bytes. Available types: None, bytes, str, bool, signed int, float, list, ndarray.

Parameters

- **data** – Data to convert.
- **as_list** – (For tests only, False by default) If False, the whole bytes message is returned. If True, the return will be a list of bytes fields.

Returns
Concatenated bytes fields (Number of fields, Size of fields, Type, Data, Args).

TcpIpClient

Bases: *TcpIpObject.TcpIpObject*

class TcpIpClient.**TcpIpClient**(*environment: Type[AbstractEnvironment], ip_address: str = 'localhost', port: int = 10000, instance_id: int = 0, instance_nb: int = 1*)

async __close() → None

Close the environment and shutdown the client.

async __communicate(*server: socket*) → None

Communication protocol with a server. First receive a command from the client, then process the appropriate actions.

Parameters
server – TcpIpServer to communicate with.

async __initialize() → None

Receive parameters from the server to create environment.

async __launch() → None

Trigger the main communication protocol with the server.

async action_on_change_db(*data: Dict[Any, Any], client_id: int, sender: socket, loop: AbstractEventLoop*) → None

Action to run when receiving the 'step' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_exit(*data: ndarray, client_id: int, sender: socket, loop: AbstractEventLoop*) → None

Action to run when receiving the 'exit' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_prediction(*data: ndarray*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'prediction' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_sample(*data: ndarray*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'sample' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_step(*data: ndarray*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'step' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

get_prediction(***kwargs*) → Dict[str, ndarray]

Request a prediction from Network.

Returns

Prediction of the Network.

initialize() → None

Receive parameters from the server to create environment.

launch() → None

Trigger the main communication protocol with the server.

request_update_visualization() → None

Triggers the Visualizer update.

TcpIpObject

class TcpIpObject.**TcpIpObject**(*ip_address: str = 'localhost', port: int = 10000*)

async __send_command(*loop: AbstractEventLoop, receiver: socket, command: str = ''*) → None

Send a bytes command among the available commands. Do not use this one. Use the dedicated function 'send_command_{cmd}(...)'.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.
- **command** – Name of the command, must be in 'self.command_dict'.

async __send_unnamed_dict(*dict_to_send: Dict[Any, Any], loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send a whole dictionary field by field as labeled data. Dictionary will be unnamed.

Parameters

- **dict_to_send** – Dictionary to send
- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

__sync_send_command(*receiver: socket, command: str = ''*) → None

Send a bytes command among the available commands. Do not use this one. Use the dedicated function 'sync_send_command_{cmd}(...)'. Synchronous version of 'TcpIpObject.send_command'.

Parameters

- **command** – Name of the command to send.
- **receiver** – TcpIpObject receiver.

__sync_send_unnamed_dict(*dict_to_send: Dict[Any, Any], receiver: Optional[socket] = None*) → None

Send a whole dictionary field by field as labeled data. Dictionary will be unnamed. Synchronous version of 'TcpIpObject.receive_labeled_data'.

Parameters

- **dict_to_send** – Dictionary to send.
- **receiver** – TcpIpObject receiver.

async action_on_change_db(*data: Dict[Any, Any], client_id: int, sender: socket, loop: AbstractEventLoop*) → None

Action to run when receiving the 'change_database' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_compute(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'compute' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_done(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'done' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_exit(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'exit' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_finished(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'finished' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_prediction(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'prediction' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.

- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_read(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'read' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_sample(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'sample' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_step(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'step' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async action_on_visualisation(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'visualisation' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async listen_while_not_done(*loop: AbstractEventLoop*, *sender: socket*, *data_dict: Dict[Any, Any]*, *client_id: Optional[int] = None*) → Dict[Any, Any]

Compute actions until 'done' command is received.

Parameters

- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.
- **data_dict** – Dictionary to collect data.
- **client_id** – ID of a Client.

async read_data(*loop: AbstractEventLoop*, *sender: socket*, *read_size: int*) → bytes

Read the data on the socket with value of buffer size as relatively small powers of 2.

Parameters

- **loop** – Asyncio event loop.
- **sender** – Socket sender.
- **read_size** – Amount of data to read on the socket.

Returns
Bytes field with 'read_size' length.

async receive_data(*loop: AbstractEventLoop*, *sender: socket*) → Union[None, bytes, str, bool, int, float, List, ndarray]

Receive data from a socket.

Parameters

- **loop** – Asyncio event loop.
- **sender** – Socket sender.

Returns
Converted data.

async receive_dict(*recv_to: Dict[Any, Any]*, *loop: Optional[AbstractEventLoop] = None*, *sender: Optional[socket] = None*) → None

Receive a whole dictionary field by field as labeled data.

Parameters

- **recv_to** – Dictionary to fill with received fields.
- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

async receive_labeled_data(*loop: AbstractEventLoop*, *sender: socket*) → Tuple[str, Union[None, bytes, str, bool, int, float, List, ndarray]]

Receive data and an associated label.

Parameters

- **loop** – Asyncio event loop.
- **sender** – TcpIpObject sender.

Returns
Label, Data.

async send_command_change_db(*loop: Optional[AbstractEventLoop] = None*, *receiver: Optional[socket] = None*) → None

Send the 'change_database' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_compute(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'compute' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_done(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'done' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_exit(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'exit' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_finished(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'finished' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_prediction(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'prediction' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_read(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'read' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_sample(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'sample' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_step(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'step' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_command_visualisation(*loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send the 'visualisation' command.

Parameters

- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_data(*data_to_send: Union[None, bytes, str, bool, int, float, List, ndarray], loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send data through the given socket.

Parameters

- **data_to_send** – Data that will be sent on socket.
- **loop** – Asyncio event loop.
- **receiver** – Socket receiver.

async send_dict(*name: str, dict_to_send: Dict[Any, Any], loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None*) → None

Send a whole dictionary field by field as labeled data.

Parameters

- **name** – Name of the dictionary.
- **dict_to_send** – Dictionary to send.
- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.

async send_labeled_data(*data_to_send: Union[None, bytes, str, bool, int, float, List, ndarray], label: str, loop: Optional[AbstractEventLoop] = None, receiver: Optional[socket] = None, send_read_command: bool = True*) → None

Send data with an associated label.

Parameters

- **data_to_send** – Data that will be sent on socket.

- **label** – Associated label.
- **loop** – Asyncio event loop.
- **receiver** – TcpIpObject receiver.
- **send_read_command** – If True, the command 'read' is sent before sending data.

sync_read_data(*read_size: int*) → bytes

Read the data on the socket with value of buffer size as relatively small powers of 2. Synchronous version of 'TcpIpObject.read_data'.

Parameters
read_size – Amount of data to read on the socket.

Returns
Bytes field with 'read_size' length.

sync_receive_data() → Union[None, bytes, str, bool, int, float, List, ndarray]

Receive data from a socket. Synchronous version of 'TcpIpObject.receive_data'.

Returns
Converted data.

sync_receive_dict(*recv_to: Dict[Any, Any], sender: Optional[socket] = None*) → None

Receive a whole dictionary field by field as labeled data. Synchronous version of 'TcpIpObject.receive_labeled_data'.

Parameters

- **recv_to** – Dictionary to fill with received fields.
- **sender** – TcpIpObject sender.

sync_receive_labeled_data() → Tuple[str, Union[None, bytes, str, bool, int, float, List, ndarray]]

Receive data and an associated label. Synchronous version of 'TcpIpObject.receive_labeled_data'.

Returns
Label, Data.

sync_send_command_change_db(*receiver: Optional[socket] = None*) → None

Send the 'change_database' command. Synchronous version of 'TcpIpObject.send_command_change_database'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_compute(*receiver: Optional[socket] = None*) → None

Send the 'compute' command. Synchronous version of 'TcpIpObject.send_command_compute'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_done(*receiver: Optional[socket] = None*) → None

Send the 'done' command. Synchronous version of 'TcpIpObject.send_command_done'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_exit(*receiver: Optional[socket] = None*) → None

Send the 'exit' command. Synchronous version of 'TcpIpObject.send_command_exit'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_finished(*receiver: Optional[socket] = None*) → None

Send the 'finished' command. Synchronous version of 'TcpIpObject.send_command_finished'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_prediction(*receiver: Optional[socket] = None*) → None

Send the 'prediction' command. Synchronous version of 'TcpIpObject.send_command_prediction'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_read(*receiver: Optional[socket] = None*) → None

Send the 'read' command. Synchronous version of 'TcpIpObject.send_command_read'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_sample(*receiver: Optional[socket] = None*) → None

Send the 'sample' command. Synchronous version of 'TcpIpObject.send_command_sample'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_step(*receiver: Optional[socket] = None*) → None

Send the 'step' command. Synchronous version of 'TcpIpObject.send_command_step'.

Parameters
receiver – TcpIpObject receiver.

sync_send_command_visualisation(*receiver: Optional[socket] = None*) → None

Send the 'visualisation' command. Synchronous version of 'TcpIpObject.send_command_visualisation'.

Parameters
receiver – TcpIpObject receiver.

sync_send_data(*data_to_send: Union[None, bytes, str, bool, int, float, List, ndarray], receiver: Optional[socket] = None*) → None

Send data through the given socket. Synchronous version of 'TcpIpObject.send_data'.

Parameters

- **data_to_send** – Data that will be sent on socket.
- **receiver** – Socket receiver.

sync_send_dict(*name: str, dict_to_send: Dict[Any, Any], receiver: Optional[socket] = None*) → None

Send a whole dictionary field by field as labeled data. Synchronous version of 'TcpIpObject.receive_labeled_data'.

Parameters

- **name** – Name of the dictionary.
- **dict_to_send** – Dictionary to send.
- **receiver** – TcpIpObject receiver.

sync_send_labeled_data(*data_to_send: Union[None, bytes, str, bool, int, float, List, ndarray]*, *label: str*, *receiver: Optional[socket] = None*, *send_read_command: bool = True*) → None

Send data with an associated label. Synchronous version of 'TcpIpObject.send_labeled_data'.

Parameters

- **data_to_send** – Data that will be sent on socket.
- **label** – Associated label.
- **receiver** – TcpIpObject receiver.
- **send_read_command** – If True, the command 'read' is sent before sending data.

TcpIpServer

Base: *TcpIpObject.TcpIpObject*

class TcpIpServer.**TcpIpServer**(*ip_address: str = 'localhost'*, *port: int = 10000*, *nb_client: int = 5*, *max_client_count: int = 10*, *batch_size: int = 5*, *manager: Optional[Any] = None*)

async __close() → None

Run server shutdown protocol.

async __communicate(*client: Optional[socket] = None*, *client_id: Optional[int] = None*, *animate: bool = True*) → None

Communication protocol with a client.

Parameters

- **client** – TcpIpObject client to communicate with.
- **client_id** – Index of the client.
- **animate** – If True, triggers an environment step.

async __connect() → None

Accept connections from clients.

__database_handler_partitions() → None

Partition update event of the DatabaseHandler.

async __initialize(*env_kwargs: Dict[str, Any]*, *visualization_db: Optional[Tuple[str, str]] = None*) → None

Send parameters to the clients to create their environments.

Parameters

- **env_kwargs** – Additional arguments to pass to the Environment.
- **visualization_db** – Path to the visualization Database to connect to.

async __request_data_to_clients(*animate: bool = True*) → None

Trigger a communication protocol for each client. Wait for all clients before to launch another communication protocol while the batch is not full.

Parameters

animate – If True, triggers an environment step

async __shutdown(*client: socket*, *idx: int*) → None

Send exit command to all clients.

Parameters

- **client** – TcpIpObject client.
- **idx** – Client index.

async action_on_prediction(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'prediction' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – asyncio.get_event_loop() return.
- **sender** – TcpIpObject sender.

async action_on_visualisation(*data: Dict[Any, Any]*, *client_id: int*, *sender: socket*, *loop: AbstractEventLoop*) → None

Action to run when receiving the 'visualisation' command.

Parameters

- **data** – Dict storing data.
- **client_id** – ID of the TcpIpClient.
- **loop** – asyncio.get_event_loop() return.
- **sender** – TcpIpObject sender.

close() → None

Run __close method with asyncio.

connect() → None

Accept connections from clients.

get_batch(*animate: bool = True*) → List[List[int]]

Build a batch from clients samples.

Parameters

animate – If True, triggers an environment step.

get_database_handler() → DatabaseHandler

Get the DatabaseHandler of the TcpIpServer.

initialize(*env_kwargs: Dict[str, Any]*, *visualization_db: Optional[Tuple[str, str]] = None*) → None

Send parameters to the clients to create their environments.

Parameters

- **env_kwargs** – Additional arguments to pass to the Environment.
- **visualization_db** – Path to the visualization Database to connect to.

set_dataset_batch(*data_lines: List[int]*) → None

Receive a batch of data from the Dataset. Samples will be dispatched between clients.

Parameters
data_lines – Batch of indices of samples.

2.10.2 Dataset

DatabaseHandler

class DatabaseHandler.**DatabaseHandler**(*on_init_handler: Optional[Callable] = None, on_partitions_handler: Optional[Callable] = None*)

add_batch(*table_name: str, batch: Dict[str, List[Any]]*) → None

Add a batch of data in a Database.

Parameters

- **table_name** – Name of the Table.
- **batch** – New lines of the Table.

add_data(*table_name: str, data: Dict[str, Any]*) → Union[int, List[int]]

Add a new line of data in a Database.

Parameters

- **table_name** – Name of the Table.
- **data** – New line of the Table.

create_fields(*table_name: str, fields: Union[List[Tuple[str, Type]], Tuple[str, Type]]*) → None

Create new Fields in a Table from one of the Databases.

Parameters

- **table_name** – Name of the Table.
- **fields** – Field or list of Fields names and types.

get_database_dir() → str

Get the database repository of the session.

get_exchange() → Database

Get the exchange Database.

get_fields(*table_name: str*) → List[str]

Get the list of Fields in a Table.

Parameters
table_name – Name of the Table.

get_line(*table_name: str, line_id: Union[int, List[int]], fields: Optional[Union[str, List[str]]] = None*) → Dict[str, Any]

Get a line of data from a Database.

Parameters

- **table_name** – Name of the Table.
- **line_id** – Index of the line to get.

- **fields** – Data fields to extract.

get_lines(*table_name: str*, *lines_id: List[List[int]]*, *fields: Optional[Union[str, List[str]]] = None*) → Dict[str, Any]

Get lines of data from a Database.

Parameters

- **table_name** – Name of the Table.
- **lines_id** – Indices of the lines to get.
- **fields** – Data fields to extract.

get_partitions() → List[Database]

Get the storing Database partitions.

init(*storing_partitions: List[Database]*, *exchange_db: Database*) → None

Initialize the list of the partitions.

Parameters

- **storing_partitions** – List of the storing Database partitions.
- **exchange_db** – Exchange Database.

init_remote(*storing_partitions: List[List[str]]*, *exchange_db: List[str]*) → None

Initialize the list of partitions in remote DatabaseHandlers.

Parameters

- **storing_partitions** – List of paths to the storing Database partitions.
- **exchange_db** – Path to the exchange Database.

load() → None

Load the Database partitions stored by the component.

update(*table_name: str*, *data: Dict[str, Any]*, *line_id: Union[int, List[int]]*) → None

Update a line in a Database.

Parameters

- **table_name** – Name of the Table.
- **data** – Updated line of the Table.
- **line_id** – Index of the line to edit.

update_list_partitions(*partition: Database*) → None

Add a new storing partition to the list.

Parameters

partition – New storing partition to add.

update_list_partitions_remote(*partition: List[str]*) → None

Add a new storing partition to the list in remote DatabaseHandler.

Parameters

partition – Path to the new storing partition.

BaseDatabaseConfig

class BaseDatabaseConfig.**BaseDatabaseConfig**(*existing_dir: Optional[str] = None, mode: Optional[str] = None, max_file_size: Optional[float] = None, shuffle: bool = False, normalize: bool = False, recompute_normalization: bool = False*)

2.10.3 Environment

BaseEnvironment

class BaseEnvironment.**BaseEnvironment**(*as_tcp_ip_client: bool = True, instance_id: int = 1, instance_nb: int = 1, **kwargs*)

apply_prediction(*prediction: Dict[str, ndarray]*) → None

Apply network prediction in environment. Not mandatory.

Parameters
prediction – Prediction data.

check_sample() → bool

Check if the current produced sample is usable for training. Not mandatory.

Returns
Current data can be used or not

close() → None

Close the Environment. Automatically called when Environment is shut down. Not mandatory.

create() → None

Create the Environment. Automatically called when Environment is launched. Must be implemented by user.

define_additional_fields(*fields: Union[List[Tuple[str, Type]], Tuple[str, Type]]*) → None

Specify the additional data fields names and types.

Parameters
fields – Field or list of Fields to tag as additional data.

define_training_fields(*fields: Union[List[Tuple[str, Type]], Tuple[str, Type]]*) → None

Specify the training data fields names and types.

Parameters
fields – Field or list of fields to tag as training data.

get_database_handler() → DatabaseHandler

Get the DatabaseHandler of the Environment.

get_prediction(***kwargs*) → Dict[str, ndarray]

Request a prediction from Network.

Returns
Network prediction.

init() → None

Initialize the Environment. Automatically called when Environment is launched. Not mandatory.

init_database() → None

Define the fields of the training dataset. Automatically called when Environment is launched. Must be implemented by user.

init_visualization() → None

Define the visualization objects to send to the Visualizer. Automatically called when Environment is launched. Not mandatory.

load_parameters() → Dict[str, Any]

Load a set of parameters from the Database.

save_parameters(**kwargs) → None

Save a set of parameters in the Database.

set_additional_data(**kwargs) → None

Set the additional data to send to the TcpIpServer or the EnvironmentManager.

set_training_data(**kwargs) → None

Set the training data to send to the TcpIpServer or the EnvironmentManager.

async step() → None

Compute the number of steps in the Environment specified by simulations_per_step in EnvironmentConfig. Must be implemented by user.

update_visualisation() → None

Triggers the Visualizer update.

BaseEnvironmentConfig

class BaseEnvironmentConfig.**BaseEnvironmentConfig**(*environment_class: Type[BaseEnvironment], as_tcp_ip_client: bool = True, number_of_thread: int = 1, ip_address: str = 'localhost', port: int = 10000, simulations_per_step: int = 1, max_wrong_samples_per_step: int = 10, load_samples: bool = False, only_first_epoch: bool = True, always_produce: bool = False, visualizer: Optional[Type[VedoVisualizer]] = None, record_wrong_samples: bool = False, env_kwargs: Optional[Dict[str, Any]] = None*)

create_environment() → BaseEnvironment

Create an Environment that will not be a TcpIpObject.

Returns

Environment object.

create_server(*environment_manager: Optional[Any] = None, batch_size: int = 1, visualization_db: Optional[Tuple[str, str]] = None*) → TcpIpServer

Create a TcpIpServer and launch TcpIpClients in subprocesses.

Parameters

- **environment_manager** – EnvironmentManager.
- **batch_size** – Number of sample in a batch.
- **visualization_db** – Path to the visualization Database to connect to.

Returns
TcpIpServer object.

`start_client`(*idx: int = 1*) → None

Run a subprocess to start a TcpIpClient.

Parameters
idx – Index of client.

`start_server`(*server: TcpIpServer, visualization_db: Optional[Tuple[str, str]] = None*) → None

Start TcpIpServer.

Parameters

- **server** – TcpIpServer.
- **visualization_db** – Path to the visualization Database to connect to.

2.10.4 Manager

DataManager

class DataManager.**DataManager**(*pipeline: Any, database_config: Optional[BaseDatabaseConfig] = None, environment_config: Optional[BaseEnvironmentConfig] = None, session: str = 'sessions/default', new_session: bool = True, produce_data: bool = True, batch_size: int = 1*)

`close`() → None

Launch the closing procedure of the DataManager.

`connect_handler`(*handler: DatabaseHandler*) → None

Add a new DatabaseHandler to the list of handlers of the DatabaseManager.

Parameters
handler – New handler to register.

`get_data`(*epoch: int = 0, animate: bool = True, load_samples: bool = True*) → None

Fetch data from the EnvironmentManager or the DatabaseManager according to the context.

Parameters

- **epoch** – Current epoch number.
- **animate** – Allow EnvironmentManager to trigger a step itself in order to generate a new sample.
- **load_samples** – If True, trigger a sample loading from the Database.

`get_prediction`(*instance_id: int*) → None

Get a Network prediction for the specified Environment instance.

`load_sample`() → List[int]

Load a sample from the Database.

Returns
Index of the loaded line.

property **nb_environment**: Optional[int]

Get the number of Environments managed by the EnvironmentManager.

property normalization: `Dict[str, List[float]]`

Get the normalization coefficients computed by the DatabaseManager.

DatabaseManager

class DatabaseManager.**DatabaseManager**(*database_config: Optional[BaseDatabaseConfig] = None, data_manager: Optional[Any] = None, pipeline: str = '', session: str = 'sessions/default', new_session: bool = True, produce_data: bool = True*)

add_data(*data_lines: Optional[List[int]] = None*) → None

Manage new lines adding in the Database.

Parameters
data_lines – Indices of the newly added lines.

change_mode(*mode: str*) → None

Change the current Database mode.

Parameters
mode – Name of the Database mode.

close()

Launch the closing procedure of the DatabaseManager.

compute_normalization() → Dict[str, List[float]]

Compute the mean and the standard deviation of all the training samples for each data field.

connect_handler(*handler: DatabaseHandler*) → None

Add and init a new DatabaseHandler to the list.

Parameters
handler – New DatabaseHandler.

create_partition() → None

Create a new partition of the Database.

get_data(*batch_size: int*) → List[List[int]]

Select a batch of indices to read in the Database.

Parameters
batch_size – Number of sample in a single batch.

get_data_shapes() → Dict[str, List[int]]

Get the shape of data Fields.

get_database_architecture() → Dict[str, List[str]]

Get the Tables and Fields structure of the Database.

get_nb_samples() → None

Get the number of sample in each partition.

get_partition_names() → List[List[str]]

Get the list of partition paths of the Database for the current mode.

get_partition_objects() → List[Database]

Get the list of partitions of the Database for the current mode.

index_samples() → None

Create a new indexing list of samples. Samples are identified by [partition_id, line_id].

load_directory(*rename_partitions: bool = False*) → None

Get the Database information from the json file (partitions, samples, etc). Load all the partitions or create one if necessary.

Parameters

rename_partitions – If True, the existing partitions should be renamed to match the session name.

static load_partitions_fields(*partition: Database*, *fields: List[str]*) → Dict[str, ndarray]

Load all the samples from a Field of a Table in the Database.

Parameters

- **partition** – Database partition to load.
- **fields** – Data Fields to get.

property normalization: Optional[Dict[str, List[float]]]

Get the normalization coefficients.

remove_empty_partitions()

Remove every empty partitions of the Database.

search_partitions_info() → None

Get the information about the Database manually if the json file is not found.

update_json() → None

Update the JSON info file with the current Database information.

update_normalization(*data_lines: List[int]*) → Dict[str, List[float]]

Update the mean and the standard deviation of all the training samples with newly added samples for each data field.

Parameters

data_lines – Indices of the newly added lines.

EnvironmentManager

class EnvironmentManager.**EnvironmentManager**(*environment_config: BaseEnvironmentConfig*, *data_manager: Optional[Any] = None*, *pipeline: str = ''*, *session: str = 'sessions/default'*, *produce_data: bool = True*, *batch_size: int = 1*)

close() → None

Launch the closing procedure of the EnvironmentManager.

update_visualizer(*instance: int*) → None

Update the Visualizer.

Parameters

instance – Index of the Environment render to update.

NetworkManager

class NetworkManager.**NetworkManager**(*network_config: BaseNetworkConfig, pipeline: str = '', session: str = 'sessions/default', new_session: bool = True*)

close() → None

Launch the closing procedure of the NetworkManager.

compute_online_prediction(*instance_id: int, normalization: Optional[Dict[str, List[float]]] = None*) → None

Make a prediction with the data passed as argument.

Parameters

- **instance_id** – Index of the Environment instance to provide a prediction.
- **normalization** – Normalization coefficients.

compute_prediction_and_loss(*optimize: bool, data_lines: List[List[int]], normalization: Optional[Dict[str, List[float]]] = None*) → Dict[str, float]

Make a prediction with the data passed as argument, optimize or not the network

Parameters

- **optimize** – If true, run a backward propagation.
- **data_lines** – Batch of indices of samples in the Database.
- **normalization** – Normalization coefficients.

Returns
The prediction and the associated loss value

get_database_handler() → DatabaseHandler

Get the DatabaseHandler of the NetworkManager.

link_clients(*nb_clients: Optional[int] = None*) → None

Update the data Exchange Database with a new line for each TcpIpClient.

Parameters
nb_clients – Number of Clients to connect.

load_network(*which_network: int = -1*) → None

Load an existing set of parameters of the Network.

Parameters
which_network – If several sets of parameters were saved, specify which one to load.

classmethod normalize_data(*data: ndarray, normalization: List[float], reverse: bool = False*) → ndarray

Apply or unapply normalization following current standard score.

Parameters

- **data** – Data to normalize.
- **normalization** – Normalization coefficients.
- **reverse** – If True, apply normalization; if False, unapply normalization.

Returns
Data with applied or misapplied normalization.

save_network(*last_save: bool = False*) → None

Save the current set of parameters of the Network.

Parameters

last_save – If True, the Network training is done then give a special denomination.

StatsManager

class StatsManager.**StatsManager**(*session: str, keep_losses: bool = False*)

add_3D_mesh(*tag: str, vertices: ndarray, colors: Optional[ndarray] = None, faces: Optional[ndarray] = None, b_n_3: bool = False, config_dict: Optional[Dict[Any, Any]] = None*) → None

Add 3D Mesh cloud to tensorboard framework.

Parameters

- **tag** (*str*) – Data identifier
- **vertices** (*ndarray*) – List of the 3D coordinates of vertices.
- **colors** (*Optional[ndarray]*) – Colors for each vertex
- **faces** (*Optional[ndarray]*) – Indices of vertices within each triangle.
- **b_n_3** (*bool*) – Data is in the format [batch_size, number_of_nodes, 3]
- **config_dict** (*Optional[Dict[Any, Any]]*) – Dictionary with ThreeJS classes names and configuration.

add_3D_point_cloud(*tag: str, vertices: ndarray, colors: Optional[ndarray] = None, b_n_3: bool = False, config_dict: Optional[Dict[Any, Any]] = None*) → None

Add 3D point cloud to tensorboard framework

Parameters

- **tag** (*str*) – Data identifier
- **vertices** (*DataContainer*) – List of the 3D coordinates of vertices.
- **colors** (*DataContainer*) – Colors for each vertex
- **b_n_3** (*bool*) – Data is in the format [batch_size, number_of_nodes, 3]
- **config_dict** (*Optional[Dict[Any, Any]]*) – Dictionary with ThreeJS classes names and configuration.

add_custom_scalar(*tag: str, value: float, count: int*) → None

Add a custom scalar to tensorboard framework.

Parameters

- **tag** (*str*) – Graph name
- **value** (*float*) – Value to store
- **count** (*int*) – ID of the value

add_custom_scalar_full(*tag: str, value: float, count: int*) → None

Add a custom scalar to tensorboard framework. Also compute mean and variance.

Parameters

- **tag** (*str*) – Graph name

- **value** (*float*) – Value to store
- **count** (*int*) – ID of the value

add_network_weight_grad(*network: Any*, *count: int*, *save_weights: bool = False*, *save_gradients: bool = True*) → None

Add network weights and gradiant if specified to tensorboard framework.

Parameters

- **network** (BaseNetwork) – Network you want to display
- **count** (*int*) – ID of the sample
- **save_weights** (*bool*) – If True will save weights to tensorboard
- **save_gradients** (*bool*) – If True will save gradient to tensorboard

add_test_loss(*value: float*, *count: int*) → None

Add test loss to tensorboard framework. Also compute mean and variance.

Parameters

- **value** (*float*) – Value to store
- **count** (*int*) – ID of the value

add_test_loss_OOB(*value: float*, *count: int*) → None

Add out of bound test loss to tensorboard framework. Also compute mean and variance.

Parameters

- **value** (*float*) – Value to store
- **count** (*int*) – ID of the value

add_train_batch_loss(*value: float*, *count: int*) → None

Add batch loss to tensorboard framework. Also compute mean and variance.

Parameters

- **value** (*float*) – Value to store
- **count** (*int*) – ID of the value

add_train_epoch_loss(*value: float*, *count: int*) → None

Add epoch loss to tensorboard framework. Also compute mean and variance.

Parameters

- **value** (*float*) – Value to store
- **count** (*int*) – ID of the value

add_train_test_batch_loss(*train_value: float*, *test_value: float*, *count: int*) → None

Add train and test batch loss to tensorboard framework.

Parameters

- **train_value** (*float*) – Value of the training batch
- **test_value** (*float*) – Value of the testing batch
- **count** (*int*) – ID of the value

add_values_multi_plot(*graph_name: str*, *tags: Iterable*, *values: Iterable*, *counts: Iterable*) → None

Plot multiples value on the same graph

Parameters

- **graph_name** (*str*) – Name of the graph
- **tags** (*Iterable*) – Iterable containing the names of the values
- **values** (*Iterable*) – Iterable containing the value
- **counts** (*Iterable*) – ID of the plots

close() → None

Launch the closing procedure of the StatsManager.

update_mean_get_var(*index: int*, *value: float*, *count: int*) → Optional[ndarray]

Update mean and return the variance of the selected value

Parameters

- **value** (*float*) – Value to add in the computation of the mean
- **index** (*int*) – Target that is updated by the value
- **count** (*int*) – ID of the value

Returns

The mean and var values over the last 50 elements

2.10.5 Network

BaseNetwork

class BaseNetwork.**BaseNetwork**(*config: namedtuple*)

forward(*input_data: Any*) → Any

Compute a forward pass of the Network.

Parameters

input_data – Input tensor.

Returns

Network prediction.

get_parameters() → Dict[str, Any]

Return the current state of Network parameters.

Returns

Network parameters.

load_parameters(*path: str*) → None

Load network parameter from path.

Parameters

path – Path to Network parameters to load.

nb_parameters() → int

Return the number of parameters of the network.

Returns

Number of parameters.

numpy_to_tensor(*data: ndarray*, *grad: bool = True*) → Any

Transform and cast data from numpy to the desired tensor type.

Parameters

- **data** – Array data to convert.
- **grad** – If True, gradient will record operations on this tensor.

Returns
Converted tensor.

predict(*data_net: Dict[str, Any]*) → Dict[str, Any]

Compute a forward pass of the Network.

Parameters
data_net – Data used by the Network.

Returns
Data produced by the Network.

save_parameters(*path: str*) → None

Saves the network parameters to the path location.

Parameters
path – Path where to save the parameters.

set_device() → None

Set computer device on which Network's parameters will be stored and tensors will be computed.

set_eval() → None

Set the Network in prediction mode (does not compute gradient).

set_train() → None

Set the Network in training mode (compute gradient).

tensor_to_numpy(*data: Any*) → ndarray

Transform and cast data from tensor type to numpy.

Parameters
data – Tensor to convert.

Returns
Converted array.

BaseNetworkConfig

class BaseNetworkConfig.**BaseNetworkConfig**(*network_class: ~typing.Type[~DeepPhysX.Core.Network.BaseNetwork.BaseNetwork] = <class 'DeepPhysX.Core.Network.BaseNetwork.BaseNetwork'>, optimization_class: ~typing.Type[~DeepPhysX.Core.Network.BaseOptimization.BaseOptimization] = <class 'DeepPhysX.Core.Network.BaseOptimization.BaseOptimization'>, data_transformation_class: ~typing.Type[~DeepPhysX.Core.Network.BaseTransformation.BaseTransformation] = <class 'DeepPhysX.Core.Network.BaseTransformation.BaseTransformation'>, network_dir: ~typing.Optional[str] = None, network_name: str = 'Network', network_type: str = 'BaseNetwork', which_network: int = -1, save_each_epoch: bool = False, data_type: str = 'float32', lr: ~typing.Optional[float] = None, require_training_stuff: bool = True, loss: ~typing.Optional[~typing.Any] = None, optimizer: ~typing.Optional[~typing.Any] = None*)

create_data_transformation() → BaseTransformation

Create an instance of data_transformation_class with given parameters.

Returns
BaseTransformation object from data_transformation_class and its parameters.

create_network() → BaseNetwork

Create an instance of network_class with given parameters.

Returns
BaseNetwork object from network_class and its parameters.

create_optimization() → BaseOptimization

Create an instance of optimization_class with given parameters.

Returns
BaseOptimization object from optimization_class and its parameters.

BaseOptimization

class BaseOptimization.**BaseOptimization**(*config: namedtuple*)

compute_loss(*data_pred: Dict[str, Any], data_opt: Dict[str, Any]*) → Dict[str, Any]

Compute loss from prediction / ground truth.

Parameters

- **data_pred** – Tensor produced by the forward pass of the Network.
- **data_opt** – Ground truth tensor to be compared with prediction.

Returns
Loss value.

optimize() → None

Run an optimization step.

set_loss() → None

Initialize the loss function.

set_optimizer(*net: BaseNetwork*) → None

Define an optimization process.

Parameters
net – Network whose parameters will be optimized.

transform_loss(*data_opt: Dict[str, Any]*) → Dict[str, float]

Apply a transformation on the loss value using the potential additional data.

Parameters
data_opt – Additional data sent as dict to compute loss value

Returns
Transformed loss value.

DataTransformation

class BaseTransformation.**BaseTransformation**(*config: namedtuple*)

transform_before_apply(*data_pred: Dict[str, Any]*) → Dict[str, Any]

Apply data operations between loss computation and prediction apply in environment.

Parameters
data_pred – Data produced by the Network.

Returns
Transformed data_pred.

transform_before_loss(*data_pred: Dict[str, Any], data_opt: Optional[Dict[str, Any]] = None*) → Tuple[Dict[str, Any], Optional[Dict[str, Any]]]

Apply data operations between network's prediction and loss computation.

Parameters

- **data_pred** – Data produced by the Network.
- **data_opt** – Data used by the Optimizer.

Returns
Transformed data_pred, data_opt.

transform_before_prediction(*data_net: Dict[str, Any]*) → Dict[str, Any]

Apply data operations before network's prediction.

Parameters
data_net – Data used by the Network.

Returns
Transformed data_net.

2.10.6 Pipelines

BaseDataGeneration

Bases: *BasePipeline.BasePipeline*

class BaseDataGeneration.**BaseDataGeneration**(*environment_config: BaseEnvironmentConfig, database_config: Optional[BaseDatabaseConfig] = None, session_dir: str = 'sessions', session_name: str = 'data_generation', new_session: bool = True, batch_nb: int = 0, batch_size: int = 0*)

batch_begin() → None

Called once at the beginning of a batch production.

batch_condition() → bool

Check the batch number condition.

batch_count() → None

Increment the batch counter.

batch_end() → None

Called once at the end of a batch production.

batch_produce() → None

Trigger the data production.

data_generation_begin() → None

Called once at the beginning of the data generation Pipeline.

data_generation_end() → None

Called once at the beginning of the data generation Pipeline.

execute() → None

Launch the data generation Pipeline. Each event is already implemented for a basic Pipeline but can also be rewritten via inheritance to describe a more complex Pipeline.

BasePipeline

class BasePipeline.**BasePipeline**(*network_config: Optional[BaseNetworkConfig] = None, database_config: Optional[BaseDatabaseConfig] = None, environment_config: Optional[BaseEnvironmentConfig] = None, session_dir: str = 'sessions', session_name: str = 'default', new_session: bool = True, pipeline: str = ''*)

execute()

Launch the Pipeline.

get_data_manager() → Optional[DataManager]

Return the DataManager associated with the Pipeline if it exists.

Returns
The DataManager associated with the Pipeline.

get_database_manager() → Optional[DatabaseManager]

Return the DatabaseManager associated with the Pipeline if it exists.

Returns
The DatabaseManager associated with the Pipeline.

get_environment_manager() → Optional[EnvironmentManager]

Return the EnvironmentManager associated with the Pipeline if it exists.

Returns
The EnvironmentManager associated with the Pipeline.

get_network_manager() → Optional[NetworkManager]

Return the NetworkManager associated with the Pipeline if it exists.

Returns
The NetworkManager associated with the Pipeline.

get_stats_manager() → Optional[StatsManager]

Return the StatsManager associated with the Pipeline if it exists.

Returns
The StatsManager associated with the Pipeline.

BasePrediction

Bases: *BasePipeline.BasePipeline*

class BasePrediction.**BasePrediction**(*network_config: BaseNetworkConfig, environment_config: BaseEnvironmentConfig, database_config: Optional[BaseDatabaseConfig] = None, session_dir: str = 'session', session_name: str = 'training', step_nb: int = -1, record: bool = False*)

execute() → None

Launch the prediction Pipeline. Each event is already implemented for a basic pipeline but can also be rewritten via inheritance to describe a more complex Pipeline.

predict() → None

Pull the data from the manager and return the prediction.

prediction_begin() → None

Called once at the beginning of the prediction Pipeline.

prediction_condition() → bool

Condition that characterize the end of the prediction Pipeline.

prediction_end() → None

Called once at the end of the prediction Pipeline.

sample_begin() → None

Called one at the beginning of each sample.

sample_end() → None

Called one at the end of each sample.

BaseTrainer

Bases: *BasePipeline.BasePipeline*

class BaseTraining.**BaseTraining**(*network_config: BaseNetworkConfig, database_config: BaseDatabaseConfig, environment_config: Optional[BaseEnvironmentConfig] = None, session_dir: str = 'sessions', session_name: str = 'training', new_session: bool = True, epoch_nb: int = 0, batch_nb: int = 0, batch_size: int = 0, debug: bool = False*)

batch_begin() → None

Called one at the beginning of a batch production.

batch_condition() → bool

Check the batch number condition.

batch_count() → None

Increment the batch counter.

batch_end() → None

Called one at the end of a batch production.

epoch_begin() → None

Called one at the beginning of each epoch.

epoch_condition() → bool

Check the epoch number condition.

epoch_count() → None

Increment the epoch counter.

epoch_end() → None

Called one at the end of each epoch.

execute() → None

Launch the training Pipeline. Each event is already implemented for a basic pipeline but can also be rewritten via inheritance to describe a more complex Pipeline.

optimize() → None

Pulls data, run a prediction and an optimizer step.

save_info_file() → None

Save a .txt file that provides a template for user notes and the description of all the components.

train_begin() → None

Called once at the beginning of the training Pipeline.

train_end() → None

Called once at the end of the training Pipeline.

2.10.7 Visualizer

VedoFactory

class VedoFactory.**VedoFactory**(*database: Optional[Database] = None, database_path: Optional[Tuple[str, str]] = None, database_dir: str = '', database_name: Optional[str] = None, remove_existing: bool = False, idx_instance: int = 0, remote: bool = False*)

render()

Render the current state of Actors in the Plotter.

VedoVisualizer

class VedoVisualizer.**VedoVisualizer**(*database: Optional[Database] = None, database_dir: str = '', database_name: Optional[str] = None, remove_existing: bool = False, offscreen: bool = False, remote: bool = False, record: bool = True*)

close()

Launch the closing procedure of the Visualizer.

render_instance(*instance: int*)

Render the current state of Actors managed by a certain Factory in the Plotter.

Parameters

instance – Index of the Environment render to update.

2.11 SOFA

Find **DeepPhysX.Sofa** API here

2.12 TORCH

Find **DeepPhysX.Torch** API here.

INDEX

Symbols

__close() (*TcpIpClient.TcpIpClient method*), 25
__close() (*TcpIpServer.TcpIpServer method*), 35
__communicate() (*TcpIpClient.TcpIpClient method*), 25
__communicate() (*TcpIpServer.TcpIpServer method*), 35
__connect() (*TcpIpServer.TcpIpServer method*), 35
__database_handler_partitions() (*TcpIpServer.TcpIpServer method*), 35
__initialize() (*TcpIpClient.TcpIpClient method*), 25
__initialize() (*TcpIpServer.TcpIpServer method*), 35
__launch() (*TcpIpClient.TcpIpClient method*), 25
__request_data_to_clients() (*TcpIpServer.TcpIpServer method*), 35
__send_command() (*TcpIpObject.TcpIpObject method*), 27
__send_unnamed_dict() (*TcpIpObject.TcpIpObject method*), 27
__shutdown() (*TcpIpServer.TcpIpServer method*), 35
__sync_send_command() (*TcpIpObject.TcpIpObject method*), 27
__sync_send_unnamed_dict() (*TcpIpObject.TcpIpObject method*), 27

A

action_on_change_db() (*TcpIpClient.TcpIpClient method*), 25
action_on_change_db() (*TcpIpObject.TcpIpObject method*), 27
action_on_compute() (*TcpIpObject.TcpIpObject method*), 27
action_on_done() (*TcpIpObject.TcpIpObject method*), 28
action_on_exit() (*TcpIpClient.TcpIpClient method*), 25
action_on_exit() (*TcpIpObject.TcpIpObject method*), 28
action_on_finished() (*TcpIpObject.TcpIpObject method*), 28
action_on_prediction() (*TcpIpClient.TcpIpClient method*), 25
action_on_prediction() (*TcpIpObject.TcpIpObject method*), 28
action_on_prediction() (*TcpIpServer.TcpIpServer method*), 36
action_on_read() (*TcpIpObject.TcpIpObject method*), 29
action_on_sample() (*TcpIpClient.TcpIpClient method*), 26
action_on_sample() (*TcpIpObject.TcpIpObject method*), 29
action_on_step() (*TcpIpClient.TcpIpClient method*), 26
action_on_step() (*TcpIpObject.TcpIpObject method*), 29
action_on_visualisation() (*TcpIpObject.TcpIpObject method*), 29
action_on_visualisation() (*TcpIpServer.TcpIpServer method*), 36
add_3D_mesh() (*StatsManager.StatsManager method*), 45
add_3D_point_cloud() (*StatsManager.StatsManager method*), 45
add_batch() (*DatabaseHandler.DatabaseHandler method*), 37
add_custom_scalar() (*StatsManager.StatsManager method*), 45
add_custom_scalar_full() (*StatsManager.StatsManager method*), 45
add_data() (*DatabaseHandler.DatabaseHandler method*), 37
add_data() (*DatabaseManager.DatabaseManager method*), 42
add_network_weight_grad() (*StatsManager.StatsManager method*), 46
add_test_loss() (*StatsManager.StatsManager method*), 46
add_test_loss_OOB() (*StatsManager.StatsManager method*), 46
add_train_batch_loss() (*StatsManager.StatsManager method*), 46
add_train_epoch_loss() (*StatsManager.StatsManager method*), 46

add_train_test_batch_loss() (*StatsManager.StatsManager method*), 46
add_values_multi_plot() (*StatsManager.StatsManager method*), 46
apply_prediction() (*BaseEnvironment.BaseEnvironment method*), 39

B

BaseDatabaseConfig (*class in BaseDatabaseConfig*), 39
BaseDataGeneration (*class in BaseDataGeneration*), 51
BaseEnvironment (*class in BaseEnvironment*), 39
BaseEnvironmentConfig (*class in BaseEnvironmentConfig*), 40
BaseNetwork (*class in BaseNetwork*), 47
BaseNetworkConfig (*class in BaseNetworkConfig*), 48
BaseOptimization (*class in BaseOptimization*), 49
BasePipeline (*class in BasePipeline*), 51
BasePrediction (*class in BasePrediction*), 52
BaseTraining (*class in BaseTraining*), 53
BaseTransformation (*class in BaseTransformation*), 50
batch_begin() (*BaseDataGeneration.BaseDataGeneration method*), 51
batch_begin() (*BaseTraining.BaseTraining method*), 53
batch_condition() (*BaseDataGeneration.BaseDataGeneration method*), 51
batch_condition() (*BaseTraining.BaseTraining method*), 53
batch_count() (*BaseDataGeneration.BaseDataGeneration method*), 51
batch_count() (*BaseTraining.BaseTraining method*), 53
batch_end() (*BaseDataGeneration.BaseDataGeneration method*), 51
batch_end() (*BaseTraining.BaseTraining method*), 53
batch_produce() (*BaseDataGeneration.BaseDataGeneration method*), 51
bytes_to_data() (*BytesConverter.BytesConverter method*), 24
BytesConverter (*class in BytesConverter*), 24

C

change_mode() (*DatabaseManager.DatabaseManager method*), 42
check_sample() (*BaseEnvironment.BaseEnvironment method*), 39
close() (*BaseEnvironment.BaseEnvironment method*), 39
close() (*DatabaseManager.DatabaseManager method*), 42
close() (*DataManager.DataManager method*), 41
close() (*EnvironmentManager.EnvironmentManager method*), 43
close() (*NetworkManager.NetworkManager method*), 44
close() (*StatsManager.StatsManager method*), 47
close() (*TcpIpServer.TcpIpServer method*), 36
close() (*VedoVisualizer.VedoVisualizer method*), 54
compute_loss() (*BaseOptimization.BaseOptimization method*), 49
compute_normalization() (*DatabaseManager.DatabaseManager method*), 42
compute_online_prediction() (*NetworkManager.NetworkManager method*), 44
compute_prediction_and_loss() (*NetworkManager.NetworkManager method*), 44
connect() (*TcpIpServer.TcpIpServer method*), 36
connect_handler() (*DatabaseManager.DatabaseManager method*), 42
connect_handler() (*DataManager.DataManager method*), 41
create() (*BaseEnvironment.BaseEnvironment method*), 39
create_data_transformation() (*BaseNetworkConfig.BaseNetworkConfig method*), 49
create_environment() (*BaseEnvironmentConfig.BaseEnvironmentConfig method*), 40
create_fields() (*DatabaseHandler.DatabaseHandler method*), 37
create_network() (*BaseNetworkConfig.BaseNetworkConfig method*), 49
create_optimization() (*BaseNetworkConfig.BaseNetworkConfig method*), 49
create_partition() (*DatabaseManager.DatabaseManager method*), 42
create_server() (*BaseEnvironmentConfig.BaseEnvironmentConfig method*), 40

D

data_generation_begin() (*BaseDataGeneration.BaseDataGeneration method*), 51
data_generation_end() (*BaseDataGeneration.BaseDataGeneration method*), 51
data_to_bytes() (*BytesConverter.BytesConverter method*), 24
DatabaseHandler (*class in DatabaseHandler*), 37
DatabaseManager (*class in DatabaseManager*), 42
DataManager (*class in DataManager*), 41
define_additional_fields() (*BaseEnvironment.BaseEnvironment method*), 39
define_training_fields() (*BaseEnvironment.BaseEnvironment method*), 39

E

EnvironmentManager (*class in EnvironmentManager*),

43
epoch_begin() (*BaseTraining.BaseTraining method*), 53
epoch_condition() (*BaseTraining.BaseTraining method*), 53
epoch_count() (*BaseTraining.BaseTraining method*), 53
epoch_end() (*BaseTraining.BaseTraining method*), 53
execute() (*BaseDataGeneration.BaseDataGeneration method*), 51
execute() (*BasePipeline.BasePipeline method*), 51
execute() (*BasePrediction.BasePrediction method*), 52
execute() (*BaseTraining.BaseTraining method*), 53

F

forward() (*BaseNetwork.BaseNetwork method*), 47

G

get_batch() (*TcpIpServer.TcpIpServer method*), 36
get_data() (*DatabaseManager.DatabaseManager method*), 42
get_data() (*DataManager.DataManager method*), 41
get_data_manager() (*BasePipeline.BasePipeline method*), 51
get_data_shapes() (*DatabaseManager.DatabaseManager method*), 42
get_database_architecture() (*DatabaseManager.DatabaseManager method*), 42
get_database_dir() (*DatabaseHandler.DatabaseHandler method*), 37
get_database_handler() (*BaseEnvironment.BaseEnvironment method*), 39
get_database_handler() (*NetworkManager.NetworkManager method*), 44
get_database_handler() (*TcpIpServer.TcpIpServer method*), 36
get_database_manager() (*BasePipeline.BasePipeline method*), 51
get_environment_manager() (*BasePipeline.BasePipeline method*), 52
get_exchange() (*DatabaseHandler.DatabaseHandler method*), 37
get_fields() (*DatabaseHandler.DatabaseHandler method*), 37
get_line() (*DatabaseHandler.DatabaseHandler method*), 37
get_lines() (*DatabaseHandler.DatabaseHandler method*), 38
get_nb_samples() (*DatabaseManager.DatabaseManager method*), 42
get_network_manager() (*BasePipeline.BasePipeline method*), 52
get_parameters() (*BaseNetwork.BaseNetwork method*), 47
get_partition_names() (*DatabaseManager.DatabaseManager method*), 42
get_partition_objects() (*DatabaseManager.DatabaseManager method*), 42
get_partitions() (*DatabaseHandler.DatabaseHandler method*), 38
get_prediction() (*BaseEnvironment.BaseEnvironment method*), 39
get_prediction() (*DataManager.DataManager method*), 41
get_prediction() (*TcpIpClient.TcpIpClient method*), 26
get_stats_manager() (*BasePipeline.BasePipeline method*), 52

I

index_samples() (*DatabaseManager.DatabaseManager method*), 42
init() (*BaseEnvironment.BaseEnvironment method*), 39
init() (*DatabaseHandler.DatabaseHandler method*), 38
init_database() (*BaseEnvironment.BaseEnvironment method*), 39
init_remote() (*DatabaseHandler.DatabaseHandler method*), 38
init_visualization() (*BaseEnvironment.BaseEnvironment method*), 40
initialize() (*TcpIpClient.TcpIpClient method*), 26
initialize() (*TcpIpServer.TcpIpServer method*), 36

L

launch() (*TcpIpClient.TcpIpClient method*), 26
link_clients() (*NetworkManager.NetworkManager method*), 44
listen_while_not_done() (*TcpIpObject.TcpIpObject method*), 29
load() (*DatabaseHandler.DatabaseHandler method*), 38
load_directory() (*DatabaseManager.DatabaseManager method*), 43
load_network() (*NetworkManager.NetworkManager method*), 44
load_parameters() (*BaseEnvironment.BaseEnvironment method*), 40
load_parameters() (*BaseNetwork.BaseNetwork method*), 47
load_partitions_fields() (*DatabaseManager.DatabaseManager static method*), 43
load_sample() (*DataManager.DataManager method*), 41

N

nb_environment (*DataManager.DataManager property*), 41

nb_parameters() (*BaseNetwork.BaseNetwork method*), 47
NetworkManager (*class in NetworkManager*), 44
normalization (*DatabaseManager.DatabaseManager property*), 43
normalization (*DataManager.DataManager property*), 41
normalize_data() (*NetworkManager.NetworkManager class method*), 44
numpy_to_tensor() (*BaseNetwork.BaseNetwork method*), 48

O

optimize() (*BaseOptimization.BaseOptimization method*), 49
optimize() (*BaseTraining.BaseTraining method*), 53

P

predict() (*BaseNetwork.BaseNetwork method*), 48
predict() (*BasePrediction.BasePrediction method*), 52
prediction_begin() (*BasePrediction.BasePrediction method*), 52
prediction_condition() (*BasePrediction.BasePrediction method*), 52
prediction_end() (*BasePrediction.BasePrediction method*), 52

R

read_data() (*TcpIpObject.TcpIpObject method*), 30
receive_data() (*TcpIpObject.TcpIpObject method*), 30
receive_dict() (*TcpIpObject.TcpIpObject method*), 30
receive_labeled_data() (*TcpIpObject.TcpIpObject method*), 30
remove_empty_partitions() (*DatabaseManager.DatabaseManager method*), 43
render() (*VedoFactory.VedoFactory method*), 54
render_instance() (*VedoVisualizer.VedoVisualizer method*), 54
request_update_visualization() (*TcpIpClient.TcpIpClient method*), 26

S

sample_begin() (*BasePrediction.BasePrediction method*), 52
sample_end() (*BasePrediction.BasePrediction method*), 52
save_info_file() (*BaseTraining.BaseTraining method*), 53
save_network() (*NetworkManager.NetworkManager method*), 44
save_parameters() (*BaseEnvironment.BaseEnvironment method*), 40
save_parameters() (*BaseNetwork.BaseNetwork method*), 48
search_partitions_info() (*DatabaseManager.DatabaseManager method*), 43
send_command_change_db() (*TcpIpObject.TcpIpObject method*), 30
send_command_compute() (*TcpIpObject.TcpIpObject method*), 31
send_command_done() (*TcpIpObject.TcpIpObject method*), 31
send_command_exit() (*TcpIpObject.TcpIpObject method*), 31
send_command_finished() (*TcpIpObject.TcpIpObject method*), 31
send_command_prediction() (*TcpIpObject.TcpIpObject method*), 31
send_command_read() (*TcpIpObject.TcpIpObject method*), 31
send_command_sample() (*TcpIpObject.TcpIpObject method*), 31
send_command_step() (*TcpIpObject.TcpIpObject method*), 32
send_command_visualisation() (*TcpIpObject.TcpIpObject method*), 32
send_data() (*TcpIpObject.TcpIpObject method*), 32
send_dict() (*TcpIpObject.TcpIpObject method*), 32
send_labeled_data() (*TcpIpObject.TcpIpObject method*), 32
set_additional_data() (*BaseEnvironment.BaseEnvironment method*), 40
set_dataset_batch() (*TcpIpServer.TcpIpServer method*), 36
set_device() (*BaseNetwork.BaseNetwork method*), 48
set_eval() (*BaseNetwork.BaseNetwork method*), 48
set_loss() (*BaseOptimization.BaseOptimization method*), 49
set_optimizer() (*BaseOptimization.BaseOptimization method*), 50
set_train() (*BaseNetwork.BaseNetwork method*), 48
set_training_data() (*BaseEnvironment.BaseEnvironment method*), 40
start_client() (*BaseEnvironmentConfig.BaseEnvironmentConfig method*), 41
start_server() (*BaseEnvironmentConfig.BaseEnvironmentConfig method*), 41
StatsManager (*class in StatsManager*), 45
step() (*BaseEnvironment.BaseEnvironment method*), 40
sync_read_data() (*TcpIpObject.TcpIpObject method*), 33
sync_receive_data() (*TcpIpObject.TcpIpObject method*), 33
sync_receive_dict() (*TcpIpObject.TcpIpObject method*), 33
sync_receive_labeled_data() (*TcpIpObject.TcpIpObject method*), 33
sync_send_command_change_db() (*TcpIpOb-*

ject.TcpIpObject method), 33
sync_send_command_compute() (*TcpIpObject.TcpIpObject method*), 33
sync_send_command_done() (*TcpIpObject.TcpIpObject method*), 33
sync_send_command_exit() (*TcpIpObject.TcpIpObject method*), 33
sync_send_command_finished() (*TcpIpObject.TcpIpObject method*), 34
sync_send_command_prediction() (*TcpIpObject.TcpIpObject method*), 34
sync_send_command_read() (*TcpIpObject.TcpIpObject method*), 34
sync_send_command_sample() (*TcpIpObject.TcpIpObject method*), 34
sync_send_command_step() (*TcpIpObject.TcpIpObject method*), 34
sync_send_command_visualisation() (*TcpIpObject.TcpIpObject method*), 34
sync_send_data() (*TcpIpObject.TcpIpObject method*), 34
sync_send_dict() (*TcpIpObject.TcpIpObject method*), 34
sync_send_labeled_data() (*TcpIpObject.TcpIpObject method*), 34

T

TcpIpClient (*class in TcpIpClient*), 25
TcpIpObject (*class in TcpIpObject*), 27
TcpIpServer (*class in TcpIpServer*), 35
tensor_to_numpy() (*BaseNetwork.BaseNetwork method*), 48
train_begin() (*BaseTraining.BaseTraining method*), 53
train_end() (*BaseTraining.BaseTraining method*), 53
transform_before_apply() (*BaseTransformation.BaseTransformation method*), 50
transform_before_loss() (*BaseTransformation.BaseTransformation method*), 50
transform_before_prediction() (*BaseTransformation.BaseTransformation method*), 50
transform_loss() (*BaseOptimization.BaseOptimization method*), 50

U

update() (*DatabaseHandler.DatabaseHandler method*), 38
update_json() (*DatabaseManager.DatabaseManager method*), 43
update_list_partitions() (*DatabaseHandler.DatabaseHandler method*), 38
update_list_partitions_remote() (*DatabaseHandler.DatabaseHandler method*), 38
update_mean_get_var() (*StatsManager.StatsManager method*), 47
update_normalization() (*DatabaseManager.DatabaseManager method*), 43
update_visualisation() (*BaseEnvironment.BaseEnvironment method*), 40
update_visualizer() (*EnvironmentManager.EnvironmentManager method*), 43

V

VedoFactory (*class in VedoFactory*), 54
VedoVisualizer (*class in VedoVisualizer*), 54